Avaluació dels LLM per la generació de codi

Carles Gallel Soler

Grau d'Enginyeria Informàtica
Intel·ligència artificial

Tutor/a de TF
Dr. Robert Clarisó Viladrosa
Professor/a responsable de l'assignatura
Dr. Xavier Baró Solé

20 de juny de 2023

Universitat Oberta
de Catalunya

FITXA DEL TREBALL FINAL

Títol del treball:	*Avaluació dels LLM per la generació de codi*
Nom de l'autor:	*Carles Gallel Soler*
Nom del consultor/a:	*Dr. Robert Clarisó Viladrosa*
Nom del PRA:	*Dr. Xavier Baró Solé*
Data de lliurament (mm/aaaa):	*06/2023*
Titulació o programa:	*Grau d'Enginyeria Informàtica*
Àrea del Treball Final:	*Intel·ligència artificial*
Idioma del treball:	*Català*
Paraules clau	*Intel·ligència artificial, ChatGPT, Generació de codi*

Resum del Treball

La generació de codi mitjançant LLM (Large Language Models) és un camp que està en auge després de la popularitat d'eines com ChatGPT o GitHub Copilot. Amb elles, un programador millora considerablement el temps que tarda a desenvolupar codi font. No obstant això, no es té certesa que el codi generat sigui correcte.

Partint d'aquest context, l'objectiu d'aquest treball és dur a terme un estudi que permeti avaluar el grau de qualitat que té el codi generat aplicant diferents metodologies.

Per poder-ho aconseguir, s'ha definit una planificació amb diverses tasques específiques. Concretament, són:

1. Estudi de l'estat de l'art, on es contextualitzi quina és la situació actual del camp de la generació de codi utilitzant LLM.
2. Disseny i desenvolupament del prototip, que permeti dur a terme totes les proves necessàries enviant consultes a l'API del LLM.
3. Anàlisi dels resultats obtinguts, on es dugui a terme un estudi comparatiu sobre tots els resultats obtinguts aplicant les diferents metodologies.

Com a resultat, s'aconseguiran unes conclusions que permetre determinar la viabilitat de l'ús d'aquestes metodologies en la generació de codi amb LLM i, a més, permetrà determinar quines són les carències actuals i les possibles

millores a desenvolupar de cara al futur.

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, this 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.

Índex

Llista de figures

1. Introducció

En els últims mesos, els LLM (*Large Language Model*) han experimentat un enorme creixement gràcies a la popularització d'eines com ChatGPT o GitHub Copilot.[1] No obstant això, aquestes intel·ligències artificials ja tenien un cert recorregut dins del camp de la computació. Concretament, la primera versió pública del LLM que utilitza ChatGPT, GPT-1, va ser publicada l'any 2018 juntament amb un *paper* on explicaven el seu funcionament.[2]

Continuant amb el cas de ChatGPT, aquest ofereix múltiples opcions segons el que sol·licita l'entrada que rep: traducció, classificació i generació de texts, cercador d'informació fins a la data de la qual té constància o, fins i tot, generació de codi, entre altres.[3] És a dir, l'eina és capaç de generar codi font segons els requisits que l'usuari sol·licita. Això inclou poder crear codi en qualsevol llenguatge i sobre qualsevol classe d'algorisme. Els LLM són especialment bons en la generació de text i, al cap i a la fi, el codi font està format per paraules amb una sintaxi i una lògica concreta.

Ara bé, segons diversos estudis[3], el codi font generat no sempre manté una correctesa suficient per poder entrar en producció de manera no supervisada i, a més, l'estil de codi utilitzat pel LLM no sempre és el més adequat. De fet, a causa d'aquesta peculiaritat, ja hi ha hagut propostes on el requereixen múltiples versions d'un mateix algorisme al LLM per tal que l'usuari pugui triar-ne la que més s'adequa a les seves necessitats. En futurs apartats se'n parlarà amb més extensió.

En qualsevol cas, queda clara quina és la tendència pel que fa a l'ús d'aquests LLM en la generació de codi. Un cop es puguin aconseguir eines deterministes, és a dir, que ofereixin molta més seguretat i precisió a l'hora de generar el codi, algunes de les tasques que actualment es deleguen a persones podran ser delegades directament al LLM. Consegüentment, és necessari dur a terme un estudi concloent sobre l'impacte que pot comportar a escala social i professional una eina com la que s'acaba de proposar sense caure en sensacionalismes.

Així, doncs, el propòsit d'aquesta recerca és avaluar la generació de codi utilitzant LLM i oferir unes conclusions que permetin determinar la fiabilitat que tenen a l'hora de generar-lo i, a més, poder determinar quina és la millora del codi generat segons les diferents metodologies que s'apliquin.

Finalment, amb l'objectiu d'oferir brevetat i claredat en tota la memòria, s'ha utilitzat el llenguatge en masculí en alguns casos. Consegüentment, quan es faci referència a una o diverses persones de manera masculina, fa referència a qualsevol persona independentment del seu gènere.

1.1. Context i justificació del Treball

Millorar el rendiment dins de qualsevol branca de l'enginyeria sempre ha estat un objectiu perseguit per tothom i l'enginyeria informàtica no se'n queda exclosa. En els últims anys s'han experimentat múltiples millores multidisciplinàries: millora i optimitzacions de metodologies aplicades, millora dels entorns de desenvolupament de programari i, més recentment, la incorporació d'intel·ligència artificial dins dels IDE (*Integrated Development Environment*).

Com ja s'ha dit a l'apartat anterior, ja existeixen diverses eines funcionals com GitHub Copilot o ChatGPT. Per tant, la incorporació d'aquestes dins d'un entorn de treball, milloraria considerablement la productivitat. Ara bé, és necessari poder garantir que l'eina utilitzada sigui determinista, ja que només d'aquesta manera es podrà assegurar que el codi generat és correcte. Així doncs, existeix una necessitat real d'avaluar la qualitat del codi generat a partir d'aquests LLM i, a més, avaluar quin és el grau de millora aplicant certes metodologies. Concretament, l'estudi utilitza la metodologia TDD (*Test Driven Development*) per buscar que la qualitat del codi generat és la desitjada.

Ara bé, s'ha de tenir en compte que la planificació s'ha de poder complir dins del termini d'un semestre. En conseqüència, s'ha decidit només avaluar un LLM en comptes de fer-ho amb diversos, ja que la càrrega de treball hauria provocat que no fos factible dur a terme el treball. A més, en el moment de la planificació del treball, OpenAI només tenia habilitada l'API del LLM gpt-3.5-turbo perquè deshabilitar l'API de Codex i l'API de GPT-4 només és accessible a través d'una llista d'espera. Així, doncs, només és viable dur a terme l'estudi pel LLM gpt-3.5-turbo.

Pel que fa al codi que es vol generar, tenint en compte que l'estat actual dels LLM no permet enviar un context amb molts caràcters, l'estudi es realitzarà amb un tipus de codi del qual tingui un mínim de coneixement. D'aquesta manera, s'ha decidit que se sol·licitarà al LLM que generi codi que representin ADT (*Abstract Data Type*) bastant utilitzats: llistes enllaçades, cues, piles, algorismes d'ordenació, etc. És a dir, se sol·licitarà que creïn una classe en Java que ofereixi el comportament de l'ADT sol·licitat.

No obstant això, com que el LLM segurament haurà estat entrenat amb biblioteques públiques que ofereixen aquests ADT, no és viable requerir-li codi com el de la biblioteca de Java. En conseqüència, s'hauria de disposar d'una definició d'aquests ADT de la qual el LLM no en tingui constància. Addicionalment, per poder valorar la seva correctesa i disposar d'una col·lecció de tests per poder enviar al LLM, s'ha decidit utilitzar la biblioteca d'ADT de l'assignatura de Disseny d'Estructura de Dades de la UOC. En ella, hi ha un conjunt ampli de diversos ADT els quals el LLM no en té coneixement.

A més, per no donar-li total llibertat i forçar-lo a implementar l'ADT d'una determinada manera, se li proporcionaran les capçaleres dels mètodes que ha d'implementar. Provocant, d'aquesta manera, que no pugui proporcionar el

mateix codi amb el qual ja ha estat entrenat. És a dir, haurà d'escriure codi nou, tot i que podrà un mínim de coneixement sobre el qual està generant. A més, per saber què ha de fer exactament cadascun dels mètodes sol·licitats, també rebrà el comentari associat a ell. Concretament, com que la biblioteca disposa de comentaris en format *javadoc*, rebrà aquesta informació per entendre com ha de generar el codi d'aquell mètode dins de tot el context de la classe.

És a dir, el LLM rebrà un esquelet amb el nom de la classe, la capçalera dels mètodes que es volen desenvolupar juntament amb el seu comentari *javadoc*. La informació que no rebrà en cap moment seran els atributs necessaris per donar el comportament esperat a l'ADT. Haurà de ser el mateix LLM qui els declari segons les necessitats que identifiqui.

Aprofundint amb el codi que generarà, també s'ha decidit que l'estudi es durà a terme **generant el codi sol·licitat** de tres maneres diferents: **sense validar amb cap mena de test**, **validant el codi amb una col·lecció de tests** que s'ha d'aportar a l'hora d'enviar la sol·licitud i, finalment, **amb una col·lecció de tests que aportarà el mateix LLM**. D'aquesta manera, es pretén analitzar amb quina metodologia ofereix una millor resposta. La qualitat d'aquesta dependrà del temps d'execució que requereix el LLM para oferir el codi, en cas de ser compilable, si és capaç de superar tots els tests, si el seu comportament és l'esperat i si és el més eficient possible. És a dir, amb quina metodologia s'ofereix un resultat que sigui fiable i, en cas que no es produeixi aquest escenari amb cap de les metodologies, que es pugui determinar amb quina s'ha obtingut millors resultats.

El següent diagrama il·lustra el funcionament de les tres metodologies gràficament.

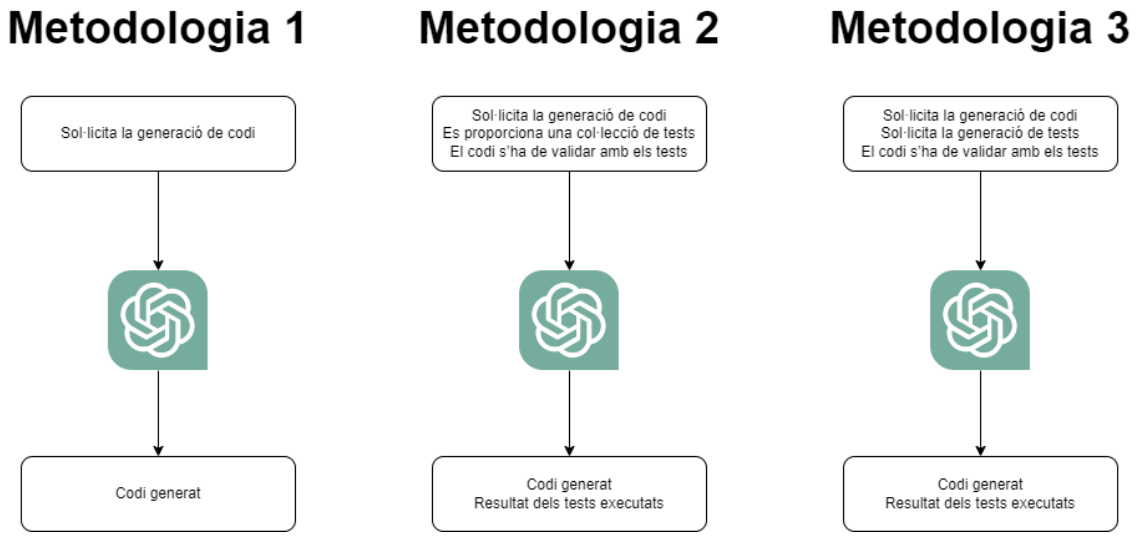


Figura 1. Diagrama que representa conceptualment el funcionament de les tres metodologies aplicades en aquest estudi. Font: original.

Tanmateix, s'ha decidit afegir altres metodologies que permetin millorar l'eficiència temporal del LLM i la correctesa del codi generat per aquest. D'aquesta manera, el prototip ha d'oferir múltiples maneres de cridar a l'API del LLM i analitzar quina d'elles és millor. En concret, per una banda, es provarà de

fer les sol·licituds amb una única crida a l'API i, per l'altra, es provarà d'utilitzar una cadena amb múltiples crides partint el *prompt* i aportant exemples sobre el que ha de fer.

Continuant amb el punt anterior, també s'ha decidit emprar *active prompting* dins del prototip per poder avaluar quina és la millora del codi generat quan s'interactua directament amb el LLM indicant-li els canvis que ha de fer sobre el primer codi font generat.

Addicionalment, partint de la premissa que en un futur els LLM podran oferir millors resultats, aprofitant el prototip que s'ha desenvolupat per a dur a terme les proves, aquest treball també té com a objectiu crear una extensió per a Visual Studio Code on es pugui enviar una sol·licitud a l'API del LLM seleccionat i afegeixi el codi font generat directament al mateix IDE.

Finalment, aprofitant la capacitat de poder usar *active prompting*, també es vol dotar a l'extensió d'aquesta funcionalitat i que es puguin indicar les modificacions necessàries des del mateix IDE.

1.2. Objectius del Treball

Com s'ha dit en apartats anteriors, l'objectiu principal d'aquest treball és dur a terme una **avaluació sobre els LLM en la generació de codi** per analitzar la seva fiabilitat. D'aquesta manera, es podran oferir unes conclusions que determinin si és viable, o no, utilitzar els LLM com a generador de codi i, a més, poder donar una resposta sobre quina és la metodologia que ofereix un millor resultat.

Formalment, aquest objectiu es descompon en els següents punts:

- Avaluació de la qualitat del codi generat amb diferents metodologies.
- Avaluació de l'eficiència temporal segons la metodologia emprada.
- Avaluació de la millora del codi iterant amb *active prompting*.

Seguidament, també ha d'oferir una conclusió sobre si és possible generar codi en entorns grans. És a dir, fins a quin punt és capaç de generar el codi en funció del context que rep.

En conseqüència, per poder garantir l'objectiu descrit, s'han de considerar les següents tasques específiques:

- Estudi de l'estat de l'art: com s'ha dit a la introducció, els LLM ja tenen un cert recorregut i, més concretament, hi ha diversos estudis on ja tracten la generació de codi utilitzant LLM. Per tant, s'ha de dur a terme un estudi sobre quines són les línies d'investigació que s'estan seguint en aquest camp, quines sembla que donen els seus fruits i quines han estat ja descartades perquè són inviables. D'aquesta manera, s'aconseguirà tenir coneixement sobre l'estat actual del camp d'estudi per poder aprofundir sobre unes metodologies o unes altres.

- Disseny i desenvolupament del prototip: per poder dur a terme totes les proves necessàries, s'ha de dissenyar i desenvolupar un prototip que permeti fer crides a l'API dels LLM estudiats aplicant totes les metodologies que es volen analitzar. Així, s'aconseguirà automatitzar el procés per poder obtenir tots els resultats necessaris de cara a l'anàlisi d'aquests.
- Anàlisi dels resultats obtinguts: un cop s'hagin llençat totes les proves i s'hagin recopilat totes les respostes necessàries, s'han d'analitzar des de diferents variables: qualitat del codi generat, temps d'espera durant la generació del codi, etc. Després, l'anàlisi ha de permetre comparar els resultats entre metodologies i poder aportar unes conclusions consistents que permetin donar resposta a l'estudi proposat com a l'objectiu principal.

En conclusió, l'objectiu d'aquest treball no pretén solucionar una problemàtica com a tal, sinó que busca avaluar si és viable, o no, i quina és la millor manera de fer-ho, el fet de generar codi a partir de llenguatge natural.

1.3. Impacte en sostenibilitat, ètic-social i de diversitat

Les intel·ligències artificials estan provocant canvis substancials de manera multidisciplinària. Concretament, tal com s'ha vist en apartats anteriors, els LLM poden comportar canvis en disciplines com l'educació, la traducció de textos i, tal com s'està observant al llarg d'aquest document, en l'enginyeria del programari. Consegüentment, s'ha de considerar l'impacte que podria arribar a tenir l'eina que s'està desenvolupant en funció dels resultats i de les conclusions obtingudes.

Des d'un punt de vista de sostenibilitat, s'ha d'analitzar tenint en compte que aquesta eina milloraria els costos temporals dels projectes, ja que el fet de poder generar el codi utilitzant un LLM escurça el temps de desenvolupament considerablement. Així, doncs, apareixen dos impactes positius: reducció de l'energia emprada per desenvolupar un projecte i reducció de costos econòmics.

El primer dels impactes citats està directament relacionat amb la programació verda, és a dir, en l'aplicació de metodologies que facin que el desenvolupament d'un programari sigui respectuós amb el medi ambient. En aquest cas ho seria per dos motius: el codi font s'hauria desenvolupat en una quantitat de temps inferior de la que s'hauria tardat desenvolupament el codi manualment i, a més, l'eina hauria d'haver usat l'algorisme més eficient possible, fet que permet fer servir la menor quantitat de recursos possibles durant la seva execució.

El segon impacte positiu té a veure amb l'àmbit econòmic. De nou, com que el programari s'hauria desenvolupat amb menys temps, els costos econòmics disminueixen considerablement, fet que la converteix en una pràctica molt atractiva per a les empreses. Ara bé, en una primera instància s'hauria d'avaluar quin impacte té l'ús d'aquesta eina sobre el cost del programari dins

del mercat, ja que podria arribar a provocar baixades de preu i fer certs productes més accessibles econòmicament parlant.

Addicionalment, continuant amb el punt de l'impacte en sostenibilitat, cal fer esment sobre la petjada ecològica que ha tingut aquest projecte. Tenint en compte que un LLM és un model preentrenat, no s'ha requerit entrenar cap model per a dur a terme l'estudi. Ara bé, com que s'han fet múltiples consultes, sí que hi ha hagut un consum d'energia. En qualsevol cas, l'impacte és extremadament inferior al que hauria estat si s'hagués hagut d'entrenar un model només pel projecte. Per tant, es pot considerar que la petjada ecològica és quasi inexistent. A més, s'ha tingut cura a l'hora de triar quines crides es feien al LLM amb l'objectiu de minimitzar aquest impacte i obtenir resultats més concloents.

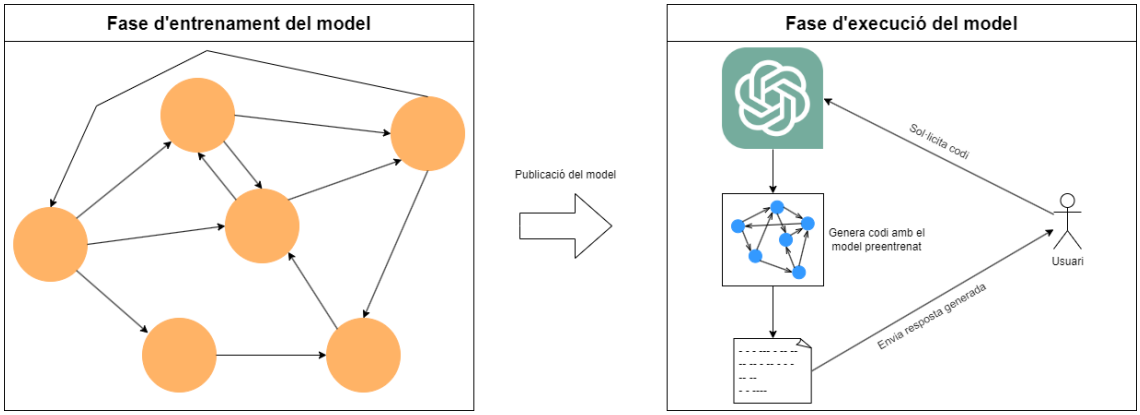


Figura 2. Diagrama conceptual que il·lustra les dues fases per les quals passa el model. La fase d'entrenament és la que provoca una petjada ecològica. Font: original.

En un àmbit ètic-social, l'ús d'aquesta eina, sempre considerant que s'ha aconseguit que sigui determinista, pot provocar que les empreses redueixen el nombre de personal dràsticament, especialment en aquells llocs de treball on la seva tasca sigui crear un codi font molt repetitiu. En canvi, si l'eina a la qual s'arriba no genera la suficient confiança, l'impacte social que tindrà serà menor i el seu ús es limitarà a acompanyar a la persona que desenvolupa el codi per fer-la més eficient. En qualsevol cas, aquest escenari apareixeria a mitjà o llarg termini, ja que s'hauria de passar per una fase d'assimilació on les empreses siguin capaces d'incorporar aquesta eina en les seves metodologies de treball. Buscar aquestes metodologies escapa de l'objectiu d'aquest treball, però és important deixar-ne constància.

Seguint dins del mateix punt, també es pot produir un impacte positiu perquè pot apropar el desenvolupament de programari a persones que abans no hi tenien accés. En conseqüència, seria més accessible i podria utilitzar-se com a porta d'entrada al camp del desenvolupament de programari. Tot i això, cal destacar que per a aquestes persones seria necessari tenir un suport en tot moment per entendre si es va pel bon camí: saber quan s'han de modificar els resultats obtinguts i saber com s'ha de relacionar el codi generat amb el qual ja existeix. Com que s'està parlant d'un cas on la persona no té coneixements avançats sobre el que està fent, aquest punt es pot arribar a relacionar amb l'ús

dels LLM dins de l'educació, ja que el mateix LLM podria ser la figura que instrueix i guia en tot moment durant el desenvolupament d'un codi font.

Tot seguit, sense abandonar l'impacte ètic-social, s'ha de destacar que no es té coneixement del conjunt d'entrenament del LLM utilitzat. Conseqüent, existeix la possibilitat que part d'aquest conjunt d'entrenament tingui llicències de propietat intel·lectual. Fet que pot comportar que el codi generat pel mateix LLM contingui fragments de codi protegits intel·lectualment i pugui acabar afectant en cas d'incloure'l dins d'un producte.

A continuació, pel que fa a l'impacte sobre la diversitat, relacionat amb el punt anterior, una eina com la mencionada permetria apropar el desenvolupament de programari a persones que no hi tenen accés actualment. Ja sigui per una situació econòmica desfavorable que impossibiliti accedir a aquest tipus de coneixements o per trencar la barrera que hi pugui haver degut a la complexitat del camp, amb una eina com aquesta es democratitzaria el desenvolupament de programari.

Finalment, cal deixar constància d'un últim punt que afecta qualsevol projecte on hi hagi una IA (intel·ligència artificial) involucrada, que correspon a la confidencialitat de les dades. En aquest cas, com que no s'està tractant amb informació confidencial, no existeix cap mena de problema relacionat amb la privadesa de dades.

1.4. Enfocament i mètode seguit

Per poder definir l'enfocament i el mètode seguit durant el projecte, s'utilitzarà el cicle de vida que defineix PMBOK[4]. Concretament, un projecte ha de constar de cinc fases:

- Iniciació: s'han d'identificar les necessitats o problemes que es volen cobrir o solucionar. A més, s'ha analitzat la viabilitat del projecte. En el cas d'aquest treball, aquesta fase s'ha donat a la PAC 0.
- Planificació: s'han d'establir i definir de manera clara quines són les fites a complir juntament amb els requisits per considerar-les completes i una estimació temporal dins d'un calendari. En aquest cas, s'han dut a terme tots aquests passos a la PAC 1.
- Execució: es la fase on s'han de completar totes les fites que s'han planificat. Acostuma a ser la fase més llarga de totes. De fet, així ha estat en aquest treball i correspon a totes les PAC des de la 2 fins a l'última (exceptuant la defensa del treball). En elles, s'han anat completant les fites planificades, incloent-hi la redacció de la memòria i la presentació del projecte.
- Seguiment i control: és la fase on s'han verificat que totes les fites s'estan complint segons la planificació i, en cas de ser necessari, aplicar accions de mitigació. Es pot considerar una fase que està present durant tot el projecte. No obstant això, les tasques de seguiment i control s'han donat al final de les fases a partir de la PAC 2 fins al final, on s'ha hagut d'actualitzar el diagrama Gantt que s'ha desenvolupat a la fase de planificació.

- Tancament: és l'última fase del projecte i és on s'han d'incloure totes les tasques per poder finalitzar el projecte degudament. Aquesta fase comparteix altres PAC que s'han classificat com a fase d'execució. Concretament, s'està parlant de la memòria, la presentació i la defensa del projecte. Si bé és cert que corresponen a la part d'execució del treball, també es poden considerar part de la fase de tancament, ja que són la fase final del treball i que permeten finalitzar el projecte, concloent amb la defensa d'aquest.

Consegüentment, tot el projecte ha seguit una metodologia en cascada tenint en compte que durant la fase d'execució, s'han hagut de fer lleugeres modificacions segons els primers resultats que s'havien obtingut.

Finalment, deixant constància del treball que es va dur a terme a la PAC 1, s'han d'esmentar possibles riscs que es van identificar inicialment. Formalment, els riscs identificables a l'inici del projecte han estat:

- Problemes tècnics a l'hora de desenvolupar el prototip. Existeix la possibilitat que apareguin dificultats tècniques durant el desenvolupament del prototip. Com a acció de mitigació es va planificar utilitzar els dies addicionals per a algunes tasques, ja que algunes de les tasques es van planificar amb diversos dies per deixar marge de maniobra.
- Falta de coneixement de les eines actuals. El desenvolupament del prototip requereix una investigació que permeti determinar quines eines i biblioteques són les més adequades per crear-lo. Això, pot comportar un biaix temporal dins de la planificació si es requereix més temps del planificat. Com a acció de mitigació es va decidir emprar part dels dies addicionals d'algunes de les tasques més simples.
- Els resultats no són els esperats. Com que el treball està orientat en dur a terme un estudi sobre la generació de codi, existeix el risc que els resultats no siguin els esperats. Per mitigar aquest risc, es va determinar que s'haurien de prioritzar els objectius i no es podrà desenvolupar el prototip. En qualsevol cas, els resultats haurien de ser analitzats i documentats a la memòria, determinant que la recerca no ha anat com s'esperava inicialment.
- Canvi en la política de serveis. L'estudi usa eines controlades per tercers, és a dir, es depèn en tot moment del manteniment del servei per part d'una empresa aliena al projecte. Consegüentment, existeix un risc elevat que el servei tanqui tenint en compte els antecedents que s'han donat, com és el cas d'Itàlia. Com a acció de mitigació, es va determinar que s'utilitzaria una VPN (*Virtual Private Network*) per poder fer les crides a l'API, en cas que el tancament sigui geopolític. Si el tancament és global, el projecte no es podrà dur a terme i no hi ha acció de mitigació possible.
- Resultats canviants segons la versió: el LLM al qual es fan les crides, GPT, té diverses actualitzacions cada cert. Aquests canvis poden provocar que els resultats obtinguts siguin diferents segons la versió. Com a acció de mitigació, s'haurà d'informar d'aquest succés i, a més,

s'hauran d'utilitzar els dies addicionals per intentar reconduir el resultat amb la nova versió.

1.5. Planificació del Treball

Per poder dur a terme la planificació d'aquest treball, s'ha tingut en compte que s'ha de poder completar dins dels terminis d'un semestre del curs acadèmic universitari. És a dir, hi ha unes dates claus marcades pel pla docent de l'assignatura. Consegüentment, s'han dividit les tasques agrupades per les entregues parcials assignades dins de l'aula.

Addicionalment, s'ha tingut en compte que la dedicació seria a temps parcial de dilluns a dissabte. Per tant, les tasques que s'han planificat contemplen un total de 300 hores de treball des de l'inici de l'estudi fins a la defensa d'aquest. Així, doncs, la planificació inicial es pot consultar a l'Annex I d'aquest document.

Com es pot veure a l'annex, la planificació està organitzada per entregues parcials on, de manera resumida, s'explicarà quines són les tasques que s'ha planificat.

- PAC 0: és la fase en la qual s'ha contextualitzat l'estudi i s'ha dut a terme una recerca sobre quin és l'estat actual del camp d'estudi proposat.
- PAC 1: fase on s'ha planificat tot el treball. Dins d'aquesta fase s'ha contemplat donar continuïtat al projecte més enllà de l'entrega final amb la redacció d'un *paper* si els resultats que s'obtenen es consideren suficients per ser publicables.
- PAC 2: s'ha organitzat de manera que sigui la primera fase de treball on es desenvolupa la base del prototip per poder dur a terme les proves i on es comencen a avaluar els primers resultats per determinar la continuïtat, o no, de la planificació inicial. És a dir, si es detecta que s'ha de dur a terme algun canvi, s'ha de fer en aquest punt. A més, és la fase on s'analitzen els resultats aconseguits aplicant les diverses metodologies TDD en la generació de codi.
- PAC 3: és la segona fase de treball on es dóna continuïtat a tot el treball fet a la fase anterior tenint en compte les possibles modificacions i s'afegeixen noves metodologies per ser estudiades. A més, és on es duen a terme totes les proves restants per poder analitzar els resultats.
- PAC 4: és la fase on s'ha planificat la redacció de la memòria i s'acaben les fites replanificades en cas que n'hi hagi.
- PAC 5a: fase on es prepararà una presentació del treball amb material audiovisual.
- PAC 5b: s'ha planificat haver de respondre les preguntes del tribunal.
- Redacció del *paper*: és una fase fora del termini del semestre universitari on, segons els resultats obtinguts, es redactarà un *paper*.

Ara bé, com que una de les tasques es va considerar opcional en tot moment, la de desenvolupar una extensió per Visual Studio Code, i no va donar temps a fer-la dins de la planificació inicial, es van dur a terme accions de mitigació amb l'objectiu de replanificar-la com una tasca de la següent entrega.

Tanmateix, després d'observar els primers resultats, es van haver de modificar algunes de les tasques que s'havien planificat perquè es va considerar que no tenia gaire sentit explorar algunes vies. Concretament, s'està parlant de la tasca on es volia aplicar la metodologia *chain-of-thought* per millorar la qualitat del codi generat. En futurs apartats s'argumentarà el perquè d'aquesta decisió amb més detall i es relacionarà amb la generació de codi creant tests propis.

Així, doncs, la planificació s'ha hagut de modificar durant el transcurs del projecte, aplicant accions de mitigació allà on ha estat necessari. La taula de fites amb la planificació final es pot consultar a l'Annex II.

Finalment, per formalitzar tota aquesta planificació i presentar-la degudament, s'ha creat un diagrama de Gantt on estan llistades cadascuna de les fites que s'han dut a terme durant el treball. Juntament amb cada fita, s'ha descrit la condició necessària per considerar-la completada. Per veure'l en detall, es pot consultar l'Annex III.

1.6. Breu sumari de productes obtinguts

Tot i ser un treball on l'objectiu principal no és crear o desenvolupar un producte final, tal com s'ha dit a l'apartat d'objectius, sí que es busca crear una eina que sigui capaç de generar el codi utilitzant l'API d'un LLM. Tanmateix, per poder dur a terme l'estudi és necessari disposar d'un prototip que permeti fer crides de manera automàtica i poder generar el codi en funció d'una entrada de text.

Consegüentment, al final d'aquest treball s'obtindrà un **prototip** funcional amb el qual es podrà generar codi mitjançant l'API d'un LLM i, a més, sigui capaç d'interactuar amb la persona sol·licitant emprant *active prompting*. El codi font del prototip es pot consultar a l'Annex IV.

De manera paral·lela, també s'aconseguiran una **col·lecció de gràfiques** que permetran entendre l'avaluació que s'està duent a terme. És a dir, a partir del codi generat, s'analitzaran els resultats i se sintetitzaran en diferents gràfiques que s'usaran al llarg de la memòria per complementar de manera gràfica els resultats assolits i les conclusions.

Finalment, a partir de gran part del codi font del prototip, també s'obtindrà una **extensió per l'IDE Visual Studio Code** que serà capaç d'afegir el codi generat, amb la metodologia que ofereix el millor resultat, dins del fitxer de codi font on s'estigui treballant segons una entrada que proporciona un usuari amb llenguatge natural.

1.7. Breu descripció dels altres capítols de la memòria

Just després de la introducció d'aquest treball, en el capítol 2, es contextualitza quin és l'estat de l'art sobre els LLM i, més concretament, sobre la generació de codi utilitzant els LLM. Addicionalment, es descriu quin és l'estat actual de les principals eines que usen LLM. A continuació, es fa una especial menció a quines són les metodologies que s'estan aplicant en la generació de codi

mitjançant un LLM. Per acabar, es parla sobre l'estat dels IDE que estan incorporant generador de codi.

És a dir, amb el capítol 2, es pretén donar un context sobre la situació actual al lector d'aquesta memòria. A més, tota la informació que s'hi troba, correspon a la síntesi de la recerca que s'ha dut a terme durant la primera fase del projecte, on s'ha hagut de contextualitzar el camp d'estudi per tenir un punt de partida.

Seguidament, al capítol 3, s'exposa el disseny del prototip utilitzat durant l'estudi. Específicament, es menciona la plataforma d'execució per a la qual està pensat, s'argumenta el perquè del llenguatge emprat, s'explica quin és el seu funcionament i com s'ha dissenyat conceptualment abans de desenvolupar-lo. Addicionalment, es mostren imatges que ajuden a entendre visualment l'arquitectura que segueix el codi del prototip.

A continuació, el següent capítol, el 4, tracta de com s'ha desenvolupat el codi per a tots els requisits i funcionalitats que ha d'oferir el prototip. A més, s'argumenta el perquè de les decisions preses en funció dels primers resultats que es van obtenir.

Pel que fa al capítol 5, és on es duu a terme una anàlisi comparativa entre tots els resultats obtinguts de manera sintetitzada i tractada. És a dir, un cop llençades totes les proves a l'API del LLM, s'ha dut a terme una anàlisi exhaustiva per analitzar cadascuna de les respostes ofertes per poder obtenir les primeres conclusions. És l'apartat on es representen els resultats de manera gràfica.

Un cop analitzats els resultats, el capítol 6 detalla la implementació de l'extensió de Visual Studio Code utilitzant la metodologia que hagi ofert millors resultats.

Per acabar, els capítols 7 i 8 ofereixen els resultats finals de tot l'estudi i les conclusions que se'n poden extreure d'aquest. Cal destacar que en aquest apartat és on s'argumenta si es donarà continuïtat, o no, a l'estudi amb la redacció d'un *paper*.

2. Estat de l’art

Durant els últims anys, els LLM han evolucionat a un ritme trepidant i, actualment, la tendència encara és la mateixa. Per tant, l'estat en el qual es trobaven els LLM ha anat variant durant el transcurs d'aquest estudi. Consegüentment, amb l'objectiu d'aportar la millor contextualització possible pel lector d'aquesta memòria, la recerca d'informació que s'ha dut a terme en aquest capítol s'ha intentat actualitzar fins a l'últim moment abans d'entregar aquest document. És a dir, s'hi han inclòs les últimes actualitzacions fins al dia 20 de juny de 2023.

2.1. Situació actual dels LLM

Gràcies als enormes avenços que hi ha hagut en els últims anys dins dels LLM, cada cop s’estan obtenint millors resultats. Cada cop, els conjunts d’entrenaments utilitzats són més i més grans, fet que permet dotar als LLM amb la capacitat de poder oferir respostes més precises segons l’entrada rebuda.[3] A més, cada cop apareixen noves eines que, aprofitant el potencial dels LLM, permeten millorar els resultats aconseguits. És a dir, s’està començant a crear una tendència on s’utilitza un LLM com a part central d’un model, tenint com a suport altres eines que li permetin millorar el resultat.

Ara bé, múltiples recerques adverteixen que existeixen algunes limitacions, com la limitació de *tokens* que existeix a l’hora d’enviar una entrada o la falta de memòria que tenen a llarg termini.

A causa del gran auge d’un LLM en concret, ChatGPT, gran part dels *papers* dels últims mesos utilitzen aquest model per a dur a terme les seves investigacions i proves. De fet, en una publicació on es fa una síntesi sobre 194 *papers* on s’analitzen o es proposen nous models o metodologies la paraula més mencionada és ChatGPT.

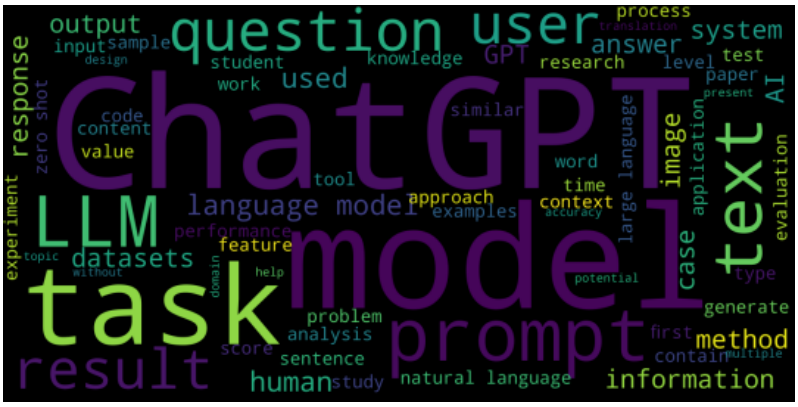


Figura 3. Núvol de paraules sobre els 194 *papers* analitzats per un estudi. Font: https://arxiv.org/pdf/2304.01852.pdf.

Actualment, els LLM comencen a explorar diversos camps on poder ser aplicats segons diversos estudis. Alguns exemples són l’educació, la traducció

de textos, la generació de texts i, tal com s'ha referenciat en tot moment durant aquest document, en la generació de codi.

Pel que fa a l'educació, pot tenir una implicació indirecta sobre aquest estudi, ja que els LLM tenen la capacitat de comprendre un codi donat o generat per ells mateixos i explicar-lo amb llenguatge natural. És a dir, aprofitant els coneixements que té, és capaç d'explicar el codi que ell mateix ha generat, en cas que sigui necessari i, a més, advertir dels possibles problemes o dificultats que poden sorgir. A més, en cas que se sol·liciti validar el codi generat amb una col·lecció de tests, com que disposa de coneixements per entendre què representa el codi generat, també ha de ser capaç de proporcionar tests que cobreixin tots els escenaris que es poden produir en un codi.

Per altra banda, la generació de texts també té implicacions directes sobre la generació de codi. Si bé és cert que el conjunt d'entrenament de la majoria dels LLM no és completament codi font, també és capaç de poder generar codi, ja que, al cap i a la fi, el codi no deixa de ser text amb una sintaxi i lògica determinada. Addicionalment, dins del camp de la traducció de texts també s'hi troba una funcionalitat que pot ser extremadament útil dins de la generació de codi. Si bé és cert que aquesta peculiaritat està més enfocada a traduir texts escrits com podria ser aquesta mateixa memòria, els LLM també tenen la capacitat de poder traduir codi d'un llenguatge de programació a un altre.

Dins de l'àmbit de la generació de codi, també hi ha hagut grans avenços amb resultats molt positius. Concretament, un estudi[6] ha pogut constatar que utilitzar tests per validar el codi generat per un LLM, com podria ser GPT-3.5, millora considerablement els resultats. A més, si aquesta metodologia es complementa amb interaccions per part de l'usuari que requereix el codi (*active prompting*), la millora és encara més notable. L'estudi mencionat ofereix quin és el grau de millora del codi generat en funció de la quantitat d'iteracions on l'usuari intervé amb indicacions per millorar-lo.

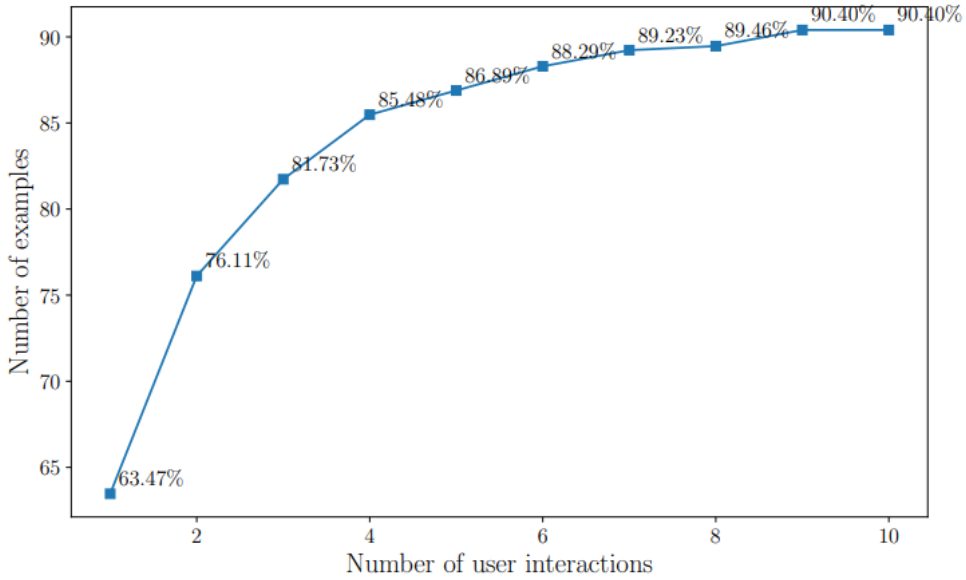


Figura 4. Percentatge de codis font que superen els tests en funció del nombre d'iteracions. Font: https://arxiv.org/pdf/2208.05950.pdf.

Conseqüentment, l'ús de tests per validar el codi generat amb indicacions per part de la persona que sol·licita el codi, ofereix uns bons resultats. No obstant això, cal destacar que cada iteració, suposa haver de fer una crida al LLM, per tant, tot i que els resultats són bons a partir d'un nombre d'iteracions, l'eficiència sembla un aspecte que ha de millorar.

Altres estudis més recents[7] també apunten a la mateixa direcció. En el *paper* citat, s'utilitza la validació mitjançant tests creats pel mateix LLM per validar el codi font generat. El seu estudi se centra a crear la millor la col·lecció de proves possible usant un nou model creat, *EvalPlus*, per poder verificar que el codi generat és correcte. És a dir, no han centrat tant el seu estudi sobre en intentar millorar el codi generat, sinó que l'han centrat sobre millorar la qualitat dels tests per poder validar amb una millor certesa la correctesa del codi. Conseqüentment, la seva recerca busca millorar el determinisme que pugui tenir la generació de codi amb un LLM. De fet, un dels problemes que apareix en la majoria dels estudis, és que no existeix una confiança cega sobre si el codi generat és correcte o no.

Continuant amb el punt anterior, sembla que aplicar TDD és una pràctica bastant habitual. Ara bé, els tests creats els ha d'executar el mateix LLM. Fet que pot comportar que no simuli correctament el codi dels tests i provoqui falsos positius, ja que un LLM no compila el codi realment. A partir d'aquesta necessitat, doncs, neix la idea de poder dotar a un LLM de la capacitat d'utilitzar altres eines, com podria ser un compilador.

Una recent publicació[5] suggereix utilitzar un LLM com a controlador d'un model, concretament ChatGPT, per poder resoldre problemes enfocats a intel·ligència artificial. És a dir, empren el mateix LLM per delegar a altres models o a ell mateix altres tasques per poder resoldre el problema inicial. Conseqüentment, la tendència que s'està buscant és dotar al LLM amb la capacitat de poder delegar a altres models per poder oferir una millor resposta.

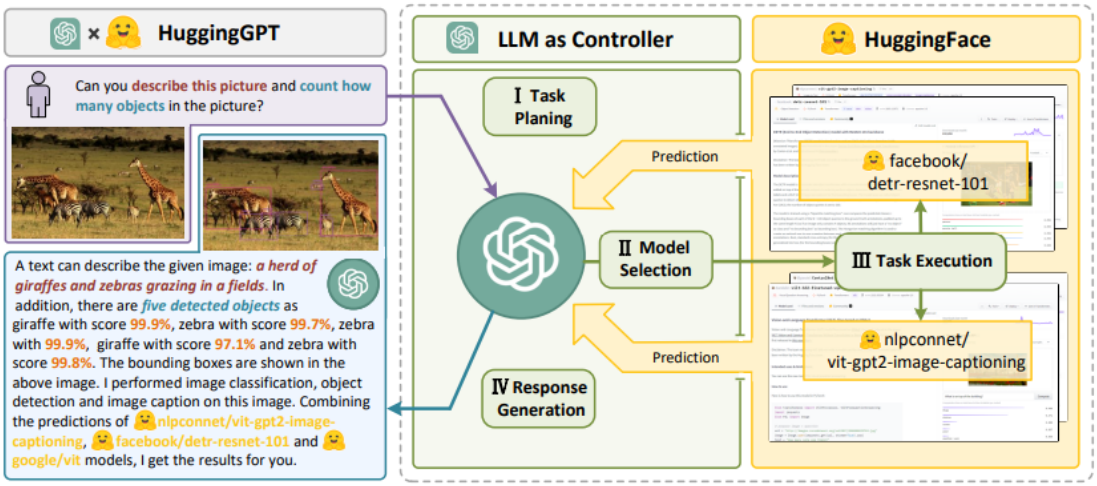


Figura 5. Esquema que representa l'ús d'un LLM com a controlador de diversos models en la tasca de delegar. Font: https://arxiv.org/pdf/2303.17580.pdf.

No obstant això, el mateix *paper* deixa clar que els resultats que s'han obtingut fins ara no han estat els desitjats, ja que existeixen algunes limitacions que també es trobaran en aquest estudi. Segons el tipus de tasca que se li demana, la dependència sobre la capacitat que té el LLM de descompondre-la en altres tasques o en la capacitat de delegar, provoca que els resultats no sempre siguin els esperats. Per aquest motiu, això provoca que el model esdevingui inestable. A més, com que s'han de dur a terme múltiples crides a diversos models, es detecten problemes d'eficiència importants. Finalment, un dels altres problemes que cita la recerca en aquest tipus de models és la limitació de *tokens* que existeix.

Tot i que en la publicació citada, no parla sobre l'aplicació d'aquesta metodologia en la generació de codi, sí que se'n pot extrapolar la idea i intentar aplicar-ho en aquest camp. Enllaçant la proposta anterior amb la generació de codi, si el LLM té la capacitat d'utilitzar un compilador per poder executar el codi amb la col·lecció de tests creats, s'aconseguiria un determinisme que abans no es tenia. A més, es podria determinar si el codi generat és compilable, o no.

En qualsevol cas, com ja s'ha dit, aquest plantejament esdevé inviable en el moment en el qual s'ha dut a terme aquest estudi degut als problemes d'eficiència que pot comportar i, fins i tot, que la limitació de *tokens* sigui un problema actualment.

Finalment, un altre estudi que aporta un punt de vista diferent en la generació de codi fent crides a un LLM presenta l'eina GPTCOMCARE[9]. Concretament, aquesta eina s'encarrega de desenvolupar *n* vegades un mateix codi font amb l'objectiu que la persona que ha sol·licitat el codi pugui triar-ne el que millor qualitat tingui. A més, utilitza colors per poder veure a simple vista quines diferències entre les diferents versions del codi hi ha.

```
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. Algorisme de l'eina GPTCOMCARE que genera *n* versions d'un mateix codi amb l'objectiu de poder-les comparar. Font: https://arxiv.org/pdf/2301.12169.pdf.

En resum, després d'analitzar múltiples estudis, els resultats obtinguts en la generació de codi són bons, però no prou per poder delegar la generació del codi a un LLM de manera no supervisada. Addicionalment, quan es delega al LLM la capacitat de poder gestionar altres models, millora considerablement els resultats. Ara bé, com que ha de dur a terme més iteracions, empitjora l'eficiència i s'arriba a uns temps d'espera considerablement llargs.

2.2. Eines actuals

Tal com s'ha dit en el punt anterior, ChatGPT és el model més popularitzat actualment. No obstant això, existeixen altres LLM que també són molt usats, fins i tot estan presents en entorns professionals, com GitHub Copilot o IntelliCode Compose. En aquest apartat, doncs, es posen en context aquestes tres eines mencionades fent èmfasi en aquelles funcionalitats i característiques més rellevants per poder entendre la seva rellevància dins del mercat mentre es comparen entre elles.

2.2.1. ChatGPT

És l'eina de la qual se n'ha sentit a parlar més en els últims mesos. Cal especificar que ChatGPT no és un LLM com a tal, sinó que és una plataforma presentada en forma de xatbot que utilitza un LLM per proporcionar respostes. Tot i això, s'ha escrit aquest terme durant la memòria com si es tractés d'un LLM per comoditat. Consegüentment, quan és tractat com un LLM es fa referència al LLM que utilitza i no a l'eina amb la conversa.

En l'inici d'aquest estudi, l'única versió del LLM que estava disponible era GPT-3.5. Actualment, disposa de diferents versions, essent la 3.5 gratuïta i, en cas de tenir una subscripció activada, es disposa de la mateixa versió 3.5, però amb una velocitat de resposta molt més elevada, del LLM GPT-4. Fins i tot, en cas d'haver aconseguit entrar a través de la llista d'espera, es disposa d'una versió que permet cercar informació per Internet, és a dir, pot arribar a tenir coneixement en temps real, i, a més, també s'ofereix una versió del mateix GPT-4, però amb *plugins*.[11]

Aquesta última funcionalitat és especialment interessant si es relaciona amb la generació de codi. Tal com s'ha dit anteriorment, els LLM poden millorar considerablement la correctesa del codi generat si són capaços d'accedir a altres models. Per tant, en l'actualitat s'ofereix la possibilitat de crear un *plugin* el qual sigui cridat just després de crear el codi i que aquest sigui compilat per un compilador real i, a més, que executi els tests que el mateix LLM ha creat.

Addicionalment, també es disposa d'una API on poder fer crides des de models externs. Lamentablement, només es poden fer crides al model gpt-3.5-turbo i, en cas d'haver accedit a través de la llista d'espera, també es pot fer ús del LLM gpt-4.

Recuperant el punt que tractava la limitació de *tokens* a l'apartat anterior, s'ha de fer una especial menció en aquesta eina. Actualment, els models entrenats permet rebre fins a 16.000 *tokens* aproximadament[8]. Just uns dies abans de l'entrega final d'aquest document es va produir un augment, dotant a les noves versions de quatre vegades més de longitud en el context. És a dir, el missatge que es pot enviar pot ser quatre vegades més gran.

Tanmateix, segons la publicació referenciada, també es permet l'opció d'utilitzar models externs. Per tant, la tendència que també segueix el mateix equip d'OpenAI és la que perseguien els *papers* que s'han referenciat anteriorment: permetre que el LLM sigui capaç de gestionar altres models segons les necessitats que consideri.

En qualsevol cas, s'ha de tenir en compte que ChatGPT és una eina que no ha estat dissenyada exclusivament per generar codi (a diferència d'altres), sinó que permet dur a terme moltes altres tasques. Per tant, no fa servir un LLM que estigui enfocat exclusivament a la generació de codi. Per una banda, aquesta característica pot semblar un problema, ja que pot donar a entendre que no té un nivell d'expertesa adequat per generar codi correctament. No obstant això, tal com es demostra en tots els estudis que s'han comentat anteriorment, aquest raonament no és correcte. Tot depèn del *prompt* que es rep, és a dir, del context que se li dóna.

2.2.2. GitHub Copilot

Aquesta eina està pensada per generar codi font, tot i que el seu conjunt d'entrenament no és només codi. El LLM que utilitza s'anomena Codex i pertany a OpenAI i GitHub. Concretament, és un LLM que parteix del model entrenat gpt-3[10]. Avui en dia, només està disponible amb la versió de pagament que ofereix GitHub o de manera gratuïta en la versió d'estudiants i professors[12].

La mateixa documentació d'aquesta eina ja dóna instruccions sobre com es pot afegir dins d'un IDE. A més, permet la generació de codi a partir de *prompts*, igual que passa amb ChatGPT, però, a més, també permet autocompletar codi de manera automàtica en funció de l'inici del codi que s'està desenvolupant. Aquesta funcionalitat no només està entrenada amb un conjunt d'entrenament extern a l'usuari final, sinó que també s'entrena a partir del codi que es desenvolupa. És a dir, té la capacitat de tenir memòria en funció del que s'ha escrit anteriorment i intentar predir si vol tornar a posar el mateix o quelcom semblant.

Actualment, la limitació de *tokens* que té aquest LLM és la meitat del que permet ChatGPT amb l'última actualització. A més, aquesta eina no permet testejar el codi de manera automàtica com sí que es pot fer amb ChatGPT. És a dir, el LLM retorna el codi generat en funció d'un *prompt*, però, en cas de voler provar si el codi és correcte, s'ha de tornar a crear un *prompt* (o utilitzar l'anterior) i especificar-li que desenvolupi tests per validar el

comportament del codi generat. Consegüentment, la validació del codi s'hauria de fer en diversos passos.

Finalment, OpenAI va deshabilitar l'accés a l'API del LLM que utilitza aquesta eina, Codex. Per tant, no es poden fer crides de manera automàtica a l'API d'aquest LLM[13].

2.2.3. IntelliCode Compose

Aquest LLM creat per Microsoft té lleugeres diferències respecte a IntelliSense. Aquest últim ofereix suggeriments d'autocompletació de codi. En canvi, IntelliCode va més enllà i fa que els suggeriments apliquin tècniques d'intel·ligència artificial en funció de com s'ha començat la línia de codi. D'aquesta manera, es dota a l'eina amb un millor percentatge de predicció i millora l'eficiència de la persona que programa.

Tot i això, cal destacar que aquesta funció no està disponible per a tots els llenguatges de programació. Concretament, aquesta eina només està disponible per C#, XAML, C++, JavaScript, TypeScript i Visual Basic.[14]

Com es pot observar, aquesta eina no ofereix la possibilitat de generar codi a partir de llenguatge natural. Per tant, és considerablement diferent de les altres dues citades anteriorment. No obstant això, considerant el seu funcionament intern i l'alt ús d'aquesta, és rellevant incloure-la dins d'aquest capítol.

Finalment, és important mencionar que aquesta eina no és accessible mitjançant alguna API i només ho és de manera gratuïta a través dels IDE oficials de Microsoft com Visual Studio i Visual Studio Code.

2.3. Metodologies d'estudi actuals

Tal com s'ha exposat al primer capítol d'aquesta memòria, aquest estudi busca avaluar la generació de codi aplicant diverses metodologies. Consegüentment, és necessari contextualitzar l'estat actual d'aquestes metodologies dins de la generació de codi, ja que, com es veurà a continuació, no només s'utilitzen per a generar codi.

2.3.1. Test Driven Development

La metodologia TDD té un llarg recorregut dins del desenvolupament de programari. Concretament, és explorada des de fa quasi 50 anys. En ella, la col·lecció de tests és la que determina la correctesa del codi desenvolupat. És a dir, es disposa d'un conjunt de proves que cobreixen tots els requisits d'un sistema o aplicació i és el codi a desenvolupar el qual s'ha de construir amb l'objectiu de superar aquestes proves.

Aquesta pràctica també s'ha vist present en diversos estudis enfocats a la generació de codi utilitzant LLM, ja que precisament permet determinar si el programari generat s'ajusta a les necessitats definides en la col·lecció de

proves. En conseqüència, pot esdevenir un element que ofereixi el determinisme que s'exigeix a qualsevol model per oferir confiança. Ara bé, si la col·lecció de tests també és generada usant un LLM, el problema no queda resolt, sinó que simplement es trasllada a una altra capa del model.

Amb l'objectiu de mitigar aquesta problemàtica, fa uns mesos es va publicar un estudi on presentaven l'eina LIBRO[15]. Aquest model empra els LLM per a la generació de tests a partir d'una entrada que proporciona un usuari. Així, quan es detecta un *bug*, aquest és reportat i introduït dins d'un *prompt* juntament amb un fitxer de tests a tall d'exemple. Un cop feta la crida al LLM amb tota aquesta entrada, aquest genera un o diversos fitxers amb tests que han de permetre cobrir tots els escenaris donats arran del bug reportat, amb l'objectiu de després poder desenvolupar el codi per superar la col·lecció de proves proporcionada pel LLM per, finalment, donar per arreglat l'error.

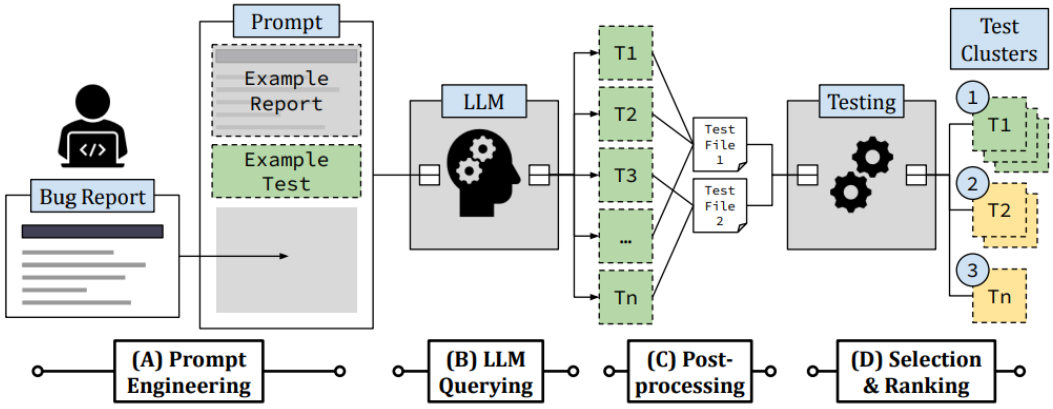


Figura 7. Representació conceptual del model LIBRO. Font: https://abhikrc.com/pdf/ICSE23.pdf.

Amb aquest model, s'ha demostrat que es mitiga la problemàtica anterior considerablement, ja que permet generar els errors a partir dels fitxers de tests amb l'objectiu que el programador els solucioni en el codi font del programa. Tot i això, l'estudi dut a terme adverteix de certs comportaments que es produeixen. Un d'aquests correspon al nombre de tests que ha de generar LLM. Conclouen que hi ha alguns *bugs* que, encara que s'augmenti considerablement el nombre de tests a generar, no necessàriament quedaran coberts. Per tant, **el nombre de tests generats no indica necessàriament que tots els requisits estan coberts**.

Fent èmfasi en el mateix apartat, un dels estudis ja citats anteriorment[7], també utilitzava aquesta metodologia per determinar si el codi generat amb el LLM és validat, o no, pels tests. Concretament, primer generen una col·lecció de tests de qualitat per després passar-los amb el codi generat. És a dir, duen a terme crides sobre el mateix LLM per generar els tests amb les dades necessàries per cobrir el màxim d'escenaris possibles. Addicionalment, també generen una col·lecció de tests més petits que garanteixin el funcionament mínim del codi amb l'objectiu de millorar l'eficiència del model. En qualsevol cas, aquest model no genera codi per ell

mateix, sinó que s'encarrega de proporcionar una col·lecció de tests en funció del codi que ja s'hagi generat anteriorment.

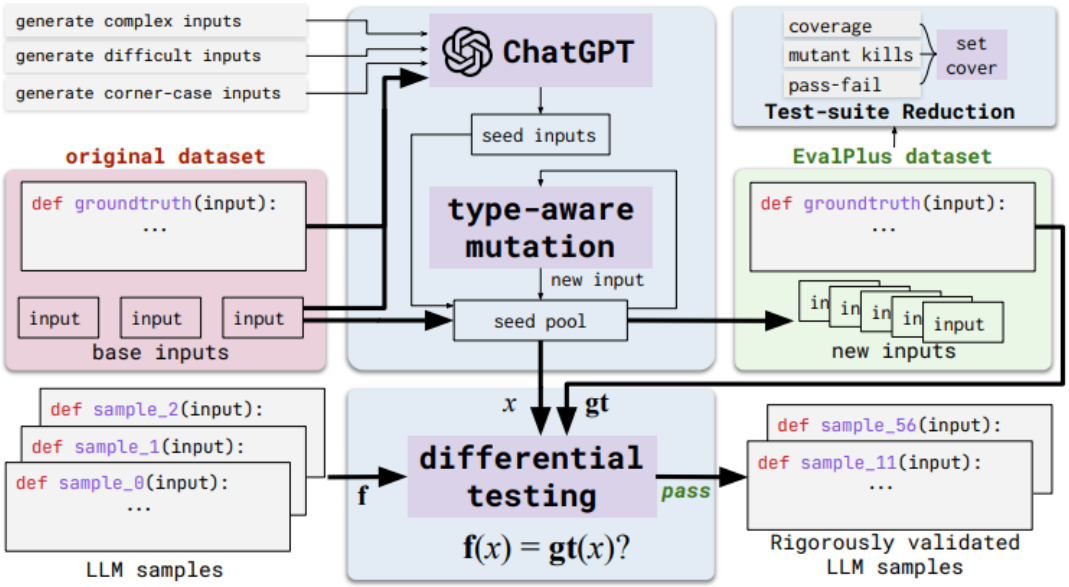


Figura 8. Representació conceptual del model *EvalPlus*, que genera una col·lecció de tests de qualitat per validar el codi generat per un LLM. Font: https://arxiv.org/abs/2305.01210.

Consegüentment, es pot afirmar que és una metodologia s'està emprant actualment com a validador en els generadors de codi i és un camí que ha de continuar tenint recorregut dins de la generació de codi amb LLM.

2.3.2. Chain-of-thought

Una de les altres metodologies que dóna molt bons resultats amb els LLM és el *chain-of-thought*. Aquesta té com a objectiu motivar el pensament crític del LLM a partir d'una cadena de pensament. És a dir, es persegueix descompondre un problema o objectiu complex amb diversos passos intermedis. D'aquesta manera, el model escriu tots aquests passos i permet generar una resposta més precisa i correcte.

Tenint en compte que un LLM és un graf amb diverses capes que determina la resposta que dóna en funció de l'entrada o del que ha escrit. Si se'l força a escriure els passos previs a la resposta final, el model és capaç de determinar amb molta més correctesa quina serà la següent paraula que ha d'utilitzar.

Aquesta metodologia ja s'ha trobat en diversos dels estudis que s'han citat. Per exemple, a l'estudi on es presentava el model LIBRO[15], es donava un fitxer de test a tall d'exemple. Si bé és cert que en aquest cas el LLM no escriu passos intermedis per arribar a la resposta, sí que ha de llegir l'entrada i analitza quin és el tipus de resposta que s'espera. Consegüentment, s'està induint un pensament crític enfocat d'una manera concreta per arribar a la resposta desitjada amb molta més precisió.

Addicionalment, també es troben estudis recents que analitzen la considerable millora d'aquesta metodologia aplicada als LLM. Per exemple, en un article publicat fa uns mesos, avaluaven la millora de les respostes donades per un LLM induint la cadena de pensament. A més, aquesta metodologia ofereix resultats considerablement bons en camps com l'aritmètica o el sentit comú. Els LLM tenen una gran capacitat de generació de text, però no apliquen un pensament lògic. Per tant, si es descomponen els problemes, la capacitat d'aplicar un pensament lògic millora. L'estudi s'ha dut a terme en diversos LLM i en tots es milloren molt els resultats.

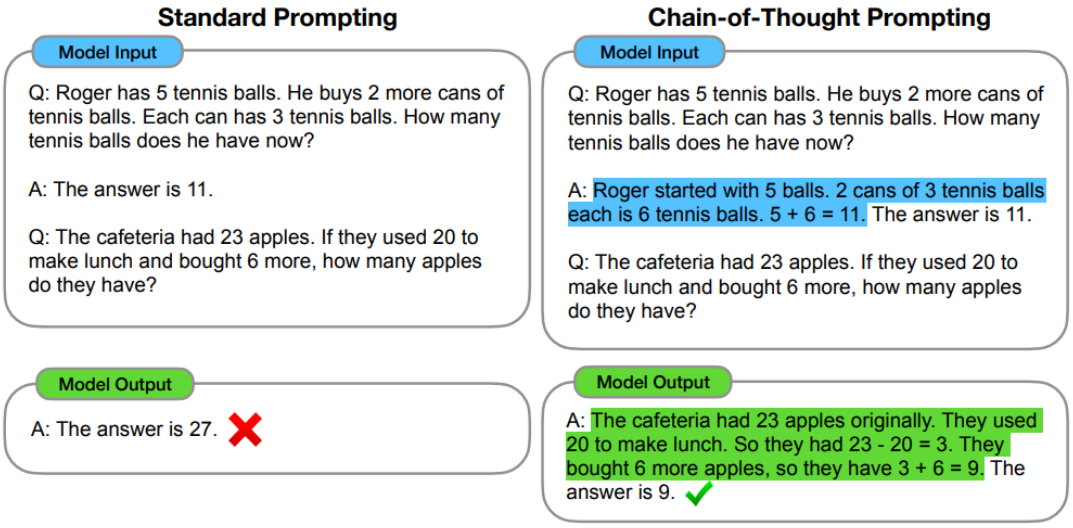


Figura 9. Comparació de les respostes entre l'aplicació i la no aplicació de la cadena de pensament. Font: https://arxiv.org/pdf/2201.11903.pdf.

En conseqüència, tenint en compte que la generació de codi conté un alt grau de pensament lògic, **l'aplicació d'aquesta metodologia sembla un element clau per obtenir millors resultats** segons els estudis mencionats.

2.3.3. *Active prompting*

Per acabar amb les metodologies rellevants en la generació de codi, és obligatori tractar l'*active prompting*. Aquesta permet interactuar amb el LLM i donar-li indicacions sobre els canvis que ha de fer sobre una primera resposta ja generada. És a dir, aquesta metodologia s'aplica un cop ja s'ha obtingut una primera resposta, s'avalua i, en cas de requerir-se, s'interactua amb el LLM.

A més, aquesta metodologia permet mitigar els possibles errors que s'hagin generat de manera manual. Tal com s'està presentant en aquest capítol, el codi generat no sempre és correcte, de manera que amb la intervenció humana es poden corregir els possibles errors o biaixos, és a dir, mitigar els problemes mencionats anteriorment. En qualsevol cas, aquesta metodologia ha d'acompanyar a les altres mencionades per poder treure'n tot el seu potencial. És a dir, aplicar *active prompting* sense una validació del codi generat amb tests, pot millorar el resultat, però és amb la combinació d'ambdues quan s'arriben a millors resultats.

Addicionalment, si aquesta metodologia també s'aplica amb la cadena de pensament mencionada anteriorment, els resultats també milloren considerablement. Un estudi recent confirma aquesta afirmació[16]. En ell, conclouen que, si els *prompts* donats per l'usuari contenen preguntes útils i es donen exemples efectius en comptes de fer preguntes arbitràries o exemples poc rellevants, els resultats obtinguts seran molt més precisos.

En conseqüència, l'aplicació d'aquesta metodologia conjuntament amb les altres ja mencionades, enriqueix el model produint respostes molt més precises i, a més, dota a un model amb la capacitat d'interactuar amb ell.

2.4. IDE amb generadors de codi

Actualment, existeixen diversos IDE que ja incorporen generadors de codi que apliquen intel·ligència artificial. De fet, en aquest capítol ja se n'han introduït alguns d'ells. Concretament, s'ha parlat del cas d'IntelliCode Compose, on s'utilitza un LLM per predir quina serà la següent paraula dins del codi d'una manera molt més precisa amb IntelliSense.

Ara bé, les eines exclusivament de Microsoft no en són l'únic exemple, ja que existeixen altres LLM com Codex que també es poden utilitzar. Tal com s'ha dit anteriorment, GitHub Copilot es pot afegir dins de Visual Studio Code mitjançant una extensió. I, de fet, aquesta no és l'única extensió que existeix perquè, com que diversos LLM permeten fer crides a l'API, qualsevol usuari pot crear la seva pròpia extensió i publicar-la. És a dir, qualsevol persona pot desenvolupar una extensió per a Visual Studio Code i fer crides a l'API del LLM que volen emprar.

Tanmateix, també existeixen altres generadors de codi incorporats dins dels mateixos IDE. Per exemple, JetBrains disposa d'un generador de codi en els seus IDE. Tot i això, aquest no fa crides a cap LLM per generar-lo, sinó que es basa en el nom de la classe i dels atributs (en cas de ser un llenguatge orientat a objectes) per generar els constructors i els *getters* i *setters* necessaris. Per tant, tot i haver-hi IDE que afirmen tenir generadors de codi, no tots utilitzen LLM per fer-ho.

La previsió és que en el futur aquesta àrea augmenti considerant amb el gran auge que estan tenint els LLM. Sigui amb els LLM que hi ha actualment, o altres de nous que en poden sortir, el gran nombre de *papers* i els bons resultats que es comencen a obtenir, fa pensar que així serà. En qualsevol cas, en tots els generadors de codi que hi ha en l'actualitat, es requereix la supervisió humana, és a dir, no hi ha cap generador de codi ara com ara que garanteixi que el codi generat és correcte. Per tant, aquestes eines s'han d'entendre com una ajuda al programador i una manera per agilitzar el procés de desenvolupament de programari. En cap cas, els estudis afirmen que, avui en dia, es pugui substituir la figura del desenvolupador de codi en entorns professionals.

3. Disseny del prototip

En aquest capítol es tractarà tot el que fa referència al disseny del prototip sense entrar en detalls sobre el desenvolupament d'aquest.

3.1. Plataforma d'execució

En una primera instància, el prototip s'ha desenvolupat pensant que la plataforma sobre la qual s'executaria seria Windows 10, és a dir, només es llançarien crides a l'API del LLM localment. Ara bé, tenint en compte que la intenció del treball és també poder oferir una extensió dins de Visual Studio Code, s'ha de permetre l'execució remotament des de qualsevol servidor en el núvol. Això implica que es poden llençar consultes al prototip allotjat en un servidor i que aquest retorni la resposta que ha donat el LLM, degudament tractada.

Així, doncs, el prototip ha de ser multiplataforma i s'ha de poder executar des de Windows 10 per poder dur a terme totes les proves necessàries i, a més, ha de poder executar-se en un servidor d'una distribució de Linux. En aquest cas, s'ha considerat que sigui una distribució de Linux perquè són els sistemes operatius amb més disponibilitat dins de les principals empreses d'allotjament. A més, la compatibilitat de les aplicacions entre les diferents versions de Linux és molt alta, per tant, es disminueix el risc que no es pugui executar degudament.

En conclusió, amb l'escenari plantejat, es cobreixen totes les necessitats que té el projecte per poder-se dur a terme satisfactòriament.

3.2. Justificació del llenguatge escollit

S'ha decidit que el llenguatge amb el qual es desenvoluparà tot el prototip sigui Python. Els motius pels quals Python és el llenguatge correcte per dur a terme aquest projecte són diversos:

- Biblioteques disponibles: Python disposa d'una gran quantitat de biblioteques públiques que permeten desenvolupar certes funcionalitats amb un temps molt inferior del que es tardaria desenvolupant-ho íntegrament. Addicionalment, moltes d'aquestes biblioteques estan enfocades justament a l'ús d'intel·ligència artificial i, més concretament, a l'ús de LLM. De fet, existeix una biblioteca basada en Python anomenada *LangChain*[17] que permet fer crides a una API d'un LLM i gestionar les respostes d'una manera molt simple. A més, també permet crear cadenes amb el mateix LLM, donant la possibilitat de mantenir una conversa amb memòria. Més endavant es parlarà detalladament d'aquesta i altres biblioteques que s'han utilitzat per desenvolupar el prototip.

- Alta compatibilitat: és just un dels requisits segons l'apartat anterior, ja que s'ha de poder executar localment (en un entorn amb Windows 10) i en un servidor al núvol. De fet, Python és un llenguatge de programació que permet instal·lar-se de manera simple a tots els entorns i plataformes d'execució que s'han mencionat anteriorment. Consegüentment, també esdevé un llenguatge adequat en aquest sentit.
- Facilitat a l'hora de treballar amb les dades: aquesta peculiaritat està relacionada amb la primera, ja que Python ofereix un gran ventall de biblioteques que permeten treballar amb diferents tipus de dades. Tot i encara no haver-ho especificat, el prototip tindrà la necessitat de gestionar les respostes rebudes en format XML. Aquesta característica del prototip es detallarà en el següent capítol, però és necessari citar-la per poder entendre la necessitat de treballar fàcilment amb diferents tipus de dades.
- Comunitat gran i activa: Python és un dels llenguatges de programació més utilitzats del món, si no el que més[18]. Gràcies a la seva simplicitat, llegibilitat, escalabilitat i compatibilitat, el converteix en una de les opcions preferides de molts desenvolupadors. Consegüentment, la comunitat vinculada a aquest llenguatge és de les més grans, fet que repercuteix positivament sobre qualsevol projecte desenvolupat en Python, ja que, en cas que sorgeixen problemes durant el desenvolupament, molt probablement es podrà trobar la solució a aquest ràpidament.

Per tant, tenint en compte tots els avantatges que ofereix desenvolupar el prototip en Python segons el context requerit pel projecte, l'elecció queda totalment justificada.

3.3. LLM utilitzats pel prototip

Tal com s'ha introduït anteriorment, l'estudi només es durà a terme utilitzant un sol LLM, concretament, gpt-3.5-turbo d'OpenAI. Tot i això, el prototip s'haurà de dissenyar mantenint una escalabilitat en aquest aspecte. És a dir, s'hauran de minimitzar els canvis a realitzar entre un LLM i un altre.

Fent èmfasi sobre la decisió d'usar un sol LLM, aquesta s'ha hagut d'adoptar per diversos motius.

El primer té a veure amb els terminis d'aquest estudi. Tenint en compte totes les metodologies que es volen avaluar, resultaria excessiu dur a terme l'estudi amb múltiples LLM considerant la planificació del projecte.

En segon lloc, reafirmant el primer dels motius, durant la planificació del treball només hi havia disponible un sol LLM a l'API d'OpenAI. Just unes setmanes abans van deshabilitar les crides a l'API de Codex i per poder accedir a l'API de gpt4 s'ha de fer mitjançant una llista d'espera, fet que produeix que la idea inicial de fer-ho amb múltiples LLM esdevingui impossible.

En resum, l'estudi només farà crides a l'API d'un sol LLM aplicant múltiples metodologies tal com s'ha explicat en el primer capítol.

3.4. Disseny conceptual del prototip

Amb l'objectiu de tenir una planificació que garanteixi una correcta escalabilitat del prototip, s'ha dut a terme un disseny conceptual d'aquest. Concretament, s'ha dissenyat el prototip en UML (*Unified Modeling Language*), on queden representades totes les metodologies que es volen aplicar. A més, també es gestiona de manera independent la lectura de la documentació en format *javadoc*, ja que es considera que és una funcionalitat que només ha de tenir el prototip amb el qual s'executen les proves.

Així, doncs, la següent figura correspon el diagrama UML que representa el prototip en un àmbit de classes.

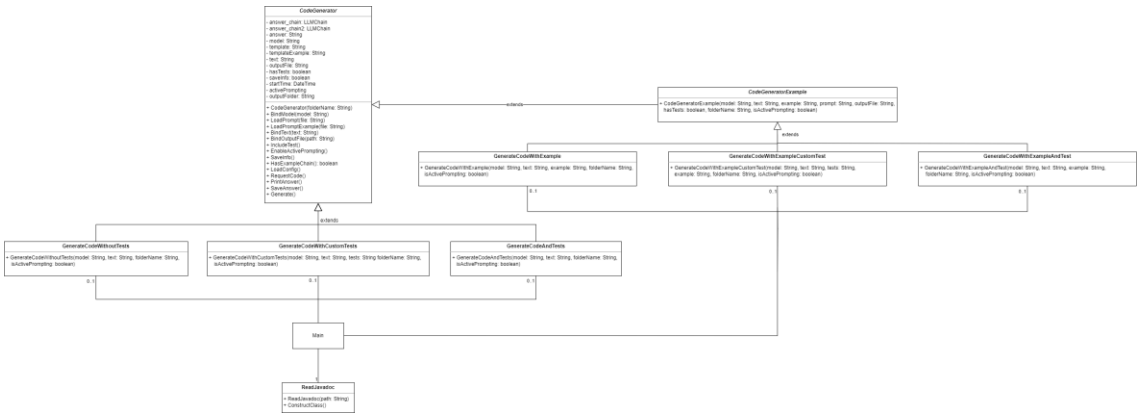


Figura 10. Diagrama UML que representa el disseny del prototip. Font: original.

Amb l'objectiu de proporcionar una millor visualització d'aquest diagrama, s'ha adjuntat un enllaç a un lloc web a l'Annex V on es pot consultar en format d'imatge.

Com es pot observar en el model proposat, hi ha una classe *CodeGenerator* que és la que s'encarrega de gestionar tot el procés de fer la crida al LLM i mostrar i emmagatzemar els resultats.

Per una banda, a partir d'aquesta classe pare, s'han creat tres classes filles instanciables que permeten generar el codi amb les tres metodologies plantejades anteriorment: sense tests, amb tests donats i amb la creació de tests pel mateix LLM.

Per l'altra, s'ha creat un nou nivell d'herència que permet generalitzar el comportament de la generació de codi utilitzant una cadena de *LangChain* on es dóna un exemple del que ha de crear abans de sol·licitar el codi. Per tant, de nou apareixen tres classes instanciables que permeten fer les crides amb diferents metodologies aportant un exemple per a cadascuna d'elles.

Pel que fa a la metodologia d'*active prompting*, s'ha dissenyat a partir d'un dels paràmetres del constructor de cadascuna de les classes instanciables.

Finalment, és necessari una classe que permeti llegir el contingut dels fitxers *javadoc* per poder enviar aquesta informació al LLM.

3.5. Execució i funcionament

Considerant que el prototip es crea per poder dur a terme una anàlisi sobre les diferents metodologies per generar codi i no és un producte final que hagi d'entrar en producció, l'execució d'aquest ha de ser la més simple possible. Consegüentment, s'ha decidit que el prototip s'executi mitjançant la línia d'ordres.

A més, com que s'han de poder executar diverses metodologies, el prototip ha d'oferir aquesta possibilitat de manera directa. Per fer-ho, s'utilitzaran diversos arguments que permetin executar el prototip d'una manera o d'una altra.

La sintaxi que s'utilitzarà per executar el prototip ha de ser la següent:

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

On:

- *1*: genera el codi sense validar-lo amb tests.
- *2*: genera el codi i el valida amb la col·lecció de tests donada.
- *3*: genera el codi i valida el codi amb tests creats per ell mateix.
- *example1*: igual que l'opció "1", però donant un exemple previ.
- *example2*: igual que l'opció "2", però donant un exemple previ.
- *example3*: igual que l'opció "3", però donant un exemple previ.
- *all*: executa l'opció 1, 2 i 3 en cadena.
- *times*: executa l'opció "all" *n* vegades. Per defecte, s'ha fixat a 10 cops.

Per tant, un exemple d'execució per la línia d'ordres hauria de ser el següent:

py __main__.py 3 active

Considerant en tot moment que s'està executant l'ordre des del directori on resideix el prototip i que el fitxer *__main__.py* conté el punt d'entrada al prototip. En aquest cas, s'estaria executant el prototip amb la generació de tests pel mateix LLM i amb la possibilitat d'aplicar *active prompting* i tornar a enviar un nou *prompt* a l'API.

En resum, tota l'execució del prototip s'ha de poder dur a terme mitjançant la línia d'ordres. Ara bé, en cas que hi hagi alguna metodologia que no ofereixi els resultats esperats, no serà necessari preparar el prototip amb l'execució d'aquesta metodologia per la línia d'ordres.

4. Desenvolupament i execució del prototip

Un cop s'ha finalitzat el disseny conceptual del prototip i s'han definit totes les necessitats d'aquest, s'ha detallat com ha estat el desenvolupament del prototip.

4.1. Preparació de l'entorn de desenvolupament

Abans de començar amb el desenvolupament, s'ha hagut de preparar l'entorn de treball amb diverses eines. Addicionalment, per poder desenvolupar el prototip, s'ha de disposar d'un IDE. En aquest cas, s'ha seleccionat Visual Studio Code per la seva comoditat, ja que ofereix una terminal integrada i permet instal·lar fàcilment diversos *plugins*.

4.1.1. Instal·lació dels llenguatges necessaris

Tal com s'ha dit anteriorment, el llenguatge de programació emprat és Python. Per tant, com que no és necessari l'ús de cap altre llenguatge per a dur a terme el desenvolupament, s'ha instal·lat una de les últimes versions estables d'aquest llenguatge, la 3.10.4.

4.1.2. Configuració amb les API

Amb l'objectiu de poder fer crides a l'API d'OpenAI, és necessari configurar una clau privada que identifica l'usuari que està duent a terme les crides a aquesta. Per fer-ho, s'ha hagut de crear una clau privada dins de la plataforma que t'ofereix OpenAI.

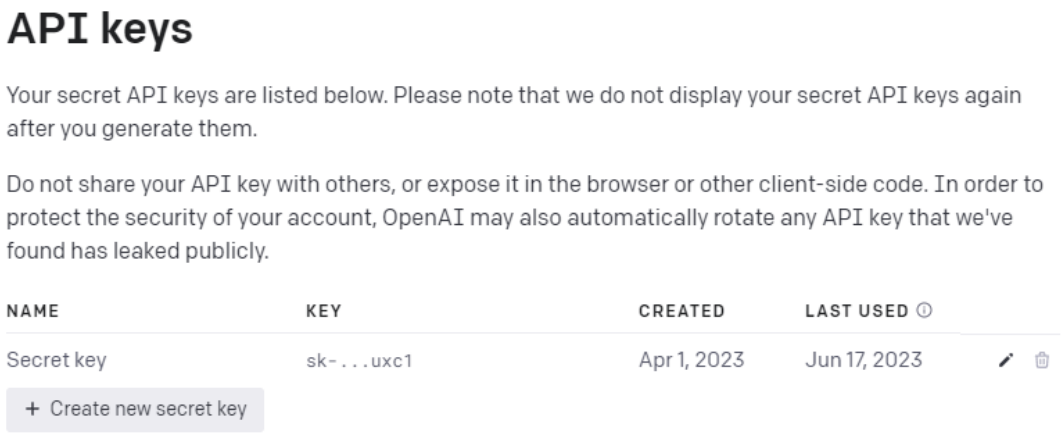


Figura 11. Captura de pantalla de la plataforma d'OpenAI on s'ha creat la clau privada. Font: original.

Un cop la clau ha estat creada, s'ha emmagatzemat a un fitxer anomenat *.env* on hi ha variables d'entorn utilitzades en el prototip. Aquest es pot consultar en el mateix repositori que s'ha adjuntat a l'Annex IV.

4.1.3. Biblioteca utilitzada per generar i validar codi

Des del principi del treball, s'ha planificat en tot moment que es treballaria amb la biblioteca LangChain, ja que ofereix mecanismes per dur a terme crides a les API de diversos LLM d'una manera molt fàcil i senzilla. Per tant, s'ha hagut d'instal·lar la biblioteca i importar-la dins del projecte.

Addicionalment, també s'ha treballat amb una altra biblioteca anomenada *GuardRails*. No obstant això, finalment s'ha hagut de descartar perquè provocava temps d'espera massa llargs en algunes situacions. Això és degut perquè la biblioteca esmentada verifica que la resposta que es rep, compleixi amb un seguit de normes definides. Consegüentment, si no les compleix, torna a sol·licitar al LLM el mateix *prompt*, de manera que s'entra en un bucle del qual pot arribar a costar sortir. Així, s'ha optat per no utilitzar-la, ja que l'eficiència temporal del prototip esdevenia considerablement inviable.

4.2. Repositori del codi font

Amb l'objectiu de facilitar el codi font desenvolupat i de poder mantenir un versionatge d'aquest, s'ha decidit crear un repositori públic per poder compartir-lo de manera ràpida, senzilla i professional. En apartats anteriors ja s'ha compartit l'enllaç d'aquest repositori. Concretament, es pot accedir a ell mitjançant l'Annex IV.

4.3. Tractament de la resposta del LLM

Com ja s'ha introduït en el capítol anterior, s'ha sol·licitat dins del prompt que el LLM retorni les respostes en llenguatge XML. D'aquesta manera, s'aconsegueix generalitzar la resposta que dóna per poder-la tractar sempre de la mateixa manera.

Concretament, s'esperen rebre dues etiquetes: *code* i *tests*. La primera ha de contenir el codi font generat pel LLM, mentre que la segona ha contenir el resultat de tots els tests generats o enviats, si és que en té. Addicionalment, cada un dels tests ha de venir dins d'una etiqueta *test* amb diferents valors com a atributs: l'identificador del test, el seu resultat, el cas que prova, el valor esperat, el valor rebut i, finalment, la raó per la qual ha fallat, si es dóna el cas.

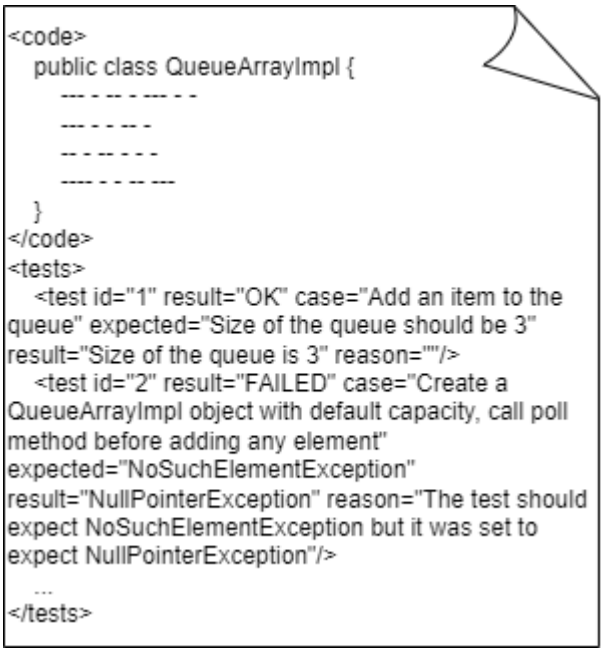


Figura 12. Exemple d'una resposta correcta amb tests per part del LLM.

Tot i haver-ho especificat clarament dins del *prompt*, inicialment es van detectar situacions on el codi generat no contenia les etiquetes necessàries per poder gestionar la resposta rebuda. Com ja s'ha dit en un apartat anterior, en un primer moment es va intentar donar solució a aquest problema amb la biblioteca *GuardRails*, però finalment es va descartar per motius d'eficiència.

Consegüentment, com es detallarà en els següents apartats, hi ha alguns casos on la resposta donada pel LLM no manté el disseny sol·licitat en el *prompt*.

Finalment, cal destacar que la resposta, sempre que aquesta retornés codi, sempre ha estat sintàcticament correcte, és a dir, no s'ha observat cap error sintàctic durant la generació. Ara bé, sí que s'han detectat errors que provocarien que no compilés el codi en algunes situacions. Per exemple, si no s'indica específicament, el LLM no inclou en algunes ocasions la inclusió de les biblioteques o classes necessàries. En qualsevol cas, aquest fet succeeix a causa de la falta de context que té sobre el conjunt del programari a generar.

4.4. Disseny dels *prompts*

Com ja s'ha anat introduint al llarg del treball, la generació de codi es duu a terme enviant un *prompt* que conté el que es vol generar utilitzant llenguatge natural. És a dir, és necessari crear un fitxer de text que contingui les indicacions per tal que el LLM sigui capaç d'entendre el que s'està sol·licitant.

Aquesta ha estat una fase on la prova i error i ha estat la metodologia emprada per arribar al resultat final. En un primer moment, el *prompt* escrit sol·licitava al LLM que generés codi com si es tractés d'un text natural. Però, després, de diverses proves i analitzar altres *papers* es va arribar a la conclusió que la millor manera per tal que un LLM entengui amb precisió què s'està sol·licitant

no és utilitzant aquest format, sinó que és a partir d'ordres clares de manera enumerada.

És a dir, s'han de definir uns requisits que ha de complir la sortida que generi el LLM. D'aquesta manera, se li està indicant que s'asseguri que la resposta compleixi tots els punts indicats.

Tot i que d'aquesta manera el resultat millorava considerablement, en algunes ocasions s'han detectat regles que no es complien. Per exemple, una de les regles és que no ha de generar comentaris dins del codi generat. I, fins i tot així, hi havia casos on afegia comentaris sobre la capçalera dels mètodes. Aquesta mena de comportaments no tenen explicació per ara, ja que fins i tot els creadors del LLM utilitzat tampoc entenen com arriben a certes conclusions.

Addicionalment, un altre aspecte totalment imprescindible que ha de tenir el *prompt* és una definició clara i concisa del rol que ha d'adoptar en tot moment. Concretament, com que la intenció és generar codi i poder-lo validar, se li ha de dir que ha d'actuar com un generador i validador de codi. És a dir, només pot proporcionar codi i els resultats dels tests, si és que en té. Aquest part és especialment important en un LLM com GPT, ja que aquest tendeix fàcilment a donar explicacions sobre quin és el procediment que segueix o quina és la justificació del que ha fet.

Relacionat amb l'aspecte anterior, si es limita el tipus de resposta que pot donar, es pot arribar a aconseguir que les respostes estiguin generalitzades, és a dir, que sempre segueixin un mateix format. A més, per poder garantir que es pugui tractar la resposta rebuda, dins del mateix *prompt* s'ha especificat en quin format ha de generar el text. En aquest cas, se li ha demanat que generi el codi dins d'una etiqueta XML anomenada *code* i que enumeri tots els tests generats dins d'una altra etiqueta anomenada *tests*. Com a resultat, la gestió de la resposta esdevé més simple perquè sempre manté el mateix format.

Tot seguit, s'ha de citar un altre fet que s'ha detectat durant el disseny dels *prompts*. Quan aquests són enviats, el LLM genera una resposta. En principi, si el *prompt* és el mateix i l'atribut *temperature* està assignat a 0, el mínim possible, hauria de retornar sempre la mateixa resposta. Sorprenentment, aquest no ha estat el cas, fent que el codi generat (i els tests) difícilment siguin els mateixos. L'escenari esmentat tindria sentit si s'hagués assignat un valor superior a 0 en la *temperature*, però existia aleatorietat en les respostes fins i tot amb aquest tipus de configuració. Per tant, tot i assignar el valor mínim a l'atribut *temperature*, sempre existeix un cert component d'aleatorietat en la generació de la resposta.

En diversos estudis ja citats en aquest document[7][15], s'han dut a terme avaluacions amb diversos valors per a aquest paràmetre, però en aquest estudi s'ha volgut posar a prova la generació de codi amb el mínim d'aleatorietat possible perquè s'entén que el component d'aleatorietat en les respostes pot ser un problema i provocar escenaris no desitjats amb més facilitat.

En resum, la millor estructura que ha de tenir un *prompt* segons les múltiples proves que s'han dut a terme, tenint en compte que és molt important el què i el com es posa, és la següent:

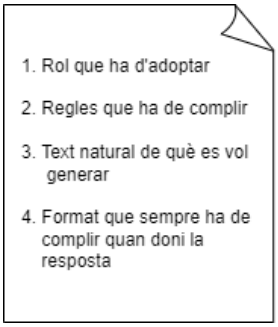


Figura 13. Esquema del contingut i l'ordre del *prompt* generat durant l'estudi. Font: original.

4.5. Generació de codi a partir de llenguatge natural

Aquesta metodologia correspon a la primera de les metodologies descrites a la introducció. Concretament, a partir del nom de la classe i de les seves classes heretades, la capçalera dels mètodes que es volen generar i el comentari associat a cadascun d'aquests mètodes, se sol·licita al LLM que generi tot el cos dels constructors i dels mètodes i declari els atributs necessaris per poder satisfer els requisits demanats en els comentaris.

Durant la generació de codi utilitzant aquesta metodologia, s'ha detectat que és quan el resultat que retorna el LLM no coincideix més vegades amb el format sol·licitat. És a dir, quan no es creen tests que validen el resultat, es provoquen situacions on, per exemple, els mètodes no tenen cos.

Ara bé, aquestes situacions s'han aconseguit mitigar modificant els *prompts*, tot i això, quan el context donat és considerablement llarg, la probabilitat que això succeeixi augmenta.

```
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 d'una part de la classe generada amb aquesta metodologia on els mètodes no tenen cos. Font: original.

Addicionalment, com es pot observar, ha afegit un comentari dins de cadascun dels mètodes com si hagués interpretat que el que es demanava no era que generés el codi, sinó que generés un esquelet sense la definició dels mètodes. Si bé és cert que aquest fet pot tenir a veure amb el contingut dels *prompts*, el fet de no tenir una validació amb tests, augmenta aquesta situació i provoca que entengui erròniament el que ha sol·licitat l'usuari.

Per tant, tot i haver modificat múltiples vegades el *prompt*, tal com es veurà en el següent capítol on es comparen els resultats, aquesta metodologia no ofereix la millor resposta. Addicionalment, com que no se sol·liciten tests, tampoc es pot oferir més informació a la persona que ha sol·licitat el codi.

En qualsevol cas, com que és la metodologia que menys text i comprovacions ha de generar, també ofereix un temps de resposta considerablement bo. Per aquest mateix motiu, és també la metodologia que ha provocat menys errors relacionats amb temps d'espera.

4.6. Generació de codi a partir de llenguatge natural i tests

A continuació, aquesta generació correspon a la metodologia número 2 que s'ha detallat al primer capítol. Per tant, és la que envia una col·lecció de tests ja creats per tal que validi el codi amb els tests enviats. Consegüentment, es converteix automàticament en la metodologia que més *tokens* necessita i, a més, la que requereix més feina per poder-la utilitzar, ja que s'han de crear els tests manualment abans d'enviar-los.

Precisament per aquest motiu, durant la generació de codi amb aquesta metodologia és quan més problemes han sorgit referents a temps d'espera.

Fins al punt que hi ha hagut classes que no s'han pogut avaluar perquè no arribaven a generar mai cap resposta. Per exemple, amb classes on només s'han de generar un nombre molt reduït de mètodes, com podria ser 4, no hi ha hagut cap problema, en canvi, quan una classe requereix diversos mètodes més, és quan es produïen els problemes.

Tanmateix, aquesta metodologia s'ha pogut aplicar en aquest cas perquè s'ha partit d'un projecte ja existent com s'ha explicat a la introducció d'aquest document. Per tant, s'ha requerit una col·lecció de tests per poder ser enviats.

Cal destacar que el LLM compila el codi generat i intenta passar els tests executant-los, si no que, a partir del text generat (el codi), determina si un test pot passar o no. Com a resultat, existeix el risc que apareguin falsos positius o falsos negatius. És a dir, quan s'han analitzat si han passat els tests, o no, hi ha hagut situacions on el LLM indicava que hi havia un test que no l'havia superat donat, però realment no era cert. Per tant, no es pot tenir la certesa que la informació que dóna respecte a la superació dels tests és totalment certa.

A més, alguns dels motius pels quals s'indica que un test ha fallat realment no haurien de suposar un problema. Això s'ha produït especialment en la gestió de les excepcions. La col·lecció de tests enviada està pensada per un codi que ha utilitzat unes excepcions en concret, però, com que el LLM pot generar el codi d'una manera diferent, pot tractar una mateixa situació amb una excepció diferent. Consegüentment, el test falla, però no és una situació on realment es trobi un error rellevant.

Finalment, s'han observat situacions que haurien d'haver-se produït amb aquesta metodologia. Tal com s'ha dit, s'ha proporcionat una col·lecció de tests amb la qual s'ha de validar el codi generat. Ara bé, quan el LLM retorna la resposta, s'han trobat diversos intents on ha afegit més tests dels que se li han proporcionat. És a dir, no ha fet cas a una part del *prompt* on se li especificava que no havia de generar tests, sinó que simplement havia de passar els que ja tenia.

El fitxer de la biblioteca que conté els tests de la classe *QueueArrayImpl*, disposa de 5 tests. Aquest és el nombre que s'ha repetit en més ocasions entre els diversos intents. No obstant això, com s'ha dit, s'han trobat diversos casos, on se n'han afegit més del compte. A més, el fet de tenir més tests dels que s'han proporcionat no ha provocat que hi hagi una millora en el codi.

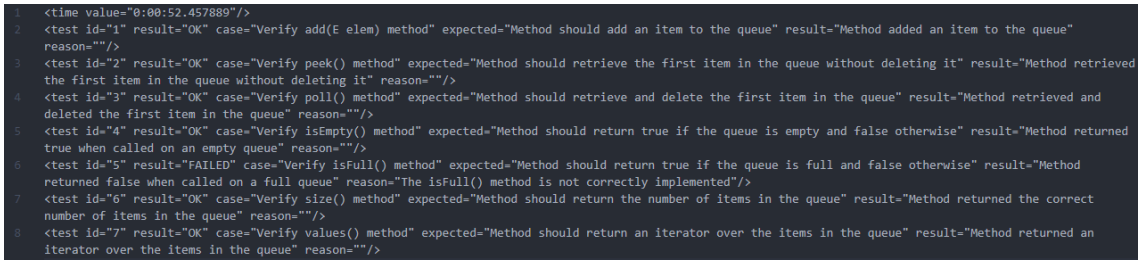


Figura 15. Captura de pantalla d'una col·lecció de tests creada corresponent a l'intent 8 de la cua. Font: original.

4.7. Generació de codi i tests a partir de llenguatge natural

Passant a la tercera de les tres metodologies explicades a l'inici del document, s'han de comentar diversos aspectes. El primer d'ells és que, tot i no enviar una quantitat molt alta de *tokens*, ja que no conté cap col·lecció de tests, es produïen temps d'espera considerablement alts. Consegüentment, tal com s'explicarà més endavant, el temps d'espera no només està associat a la quantitat de *tokens* enviats, sinó que també ho està al que es demana. Així, durant la generació de codi amb aquesta metodologia, s'han patit problemes relacionats amb els *timeouts*.

Tot i això, el codi generat que s'ha obtingut tenia un comportament molt similar, pel que fa a la correctesa, al codi original amb el qual s'ha comparat (el de la biblioteca utilitzada). Aquest succés està altament relacionat amb el concepte de *chain-of-thought*. Com que el LLM és qui ha de generar els tests per validar el codi, se'l dota amb la capacitat de proporcionar codi més precís, ja que és capaç de descompondre el requisit sol·licitat per l'usuari en aspectes més concrets.

Per exemple, com que ha de generar els tests per diversos escenaris, el codi generat té més possibilitats de contemplar aquest escenari. És a dir, si determina que hi ha d'haver un test que controli si un vector està buit, el codi contindrà el control d'aquest escenari.

Per altra banda, s'ha observat que el nombre de tests, tot i utilitzar exactament el mateix *prompt* i assignar l'atribut de la *temperature* a 0, sempre va variant i difícilment repeteix el mateix tipus de tests. A continuació es mostra un exemple de dues versions de tests generats quan se li sol·licitava una llista enllaçada:

```
<time value="0:00:26.392807"/>
<test id="1" result="OK" case="Test LinkedList constructor" expected="true" result="true" reason=""/>
<test id="2" result="OK" case="Test delete method" expected="true" result="true" reason=""/>
<test id="3" result="OK" case="Test deleteFirst method" expected="true" result="true" reason=""/>
<test id="4" result="OK" case="Test deleteNext method" expected="true" result="true" reason=""/>
<test id="5" result="OK" case="Test insertAfter method" expected="true" result="true" reason=""/>
<test id="6" result="OK" case="Test insertBefore method" expected="true" result="true" reason=""/>
<test id="7" result="OK" case="Test insertBeginning method" expected="true" result="true" reason=""/>
<test id="8" result="OK" case="Test insertEnd method" expected="true" result="true" reason=""/>
<test id="9" result="OK" case="Test isEmpty method" expected="true" result="true" reason=""/>
<test id="10" result="OK" case="Test newPosition method" expected="true" result="true" reason=""/>
<test id="11" result="OK" case="Test positions method" expected="true" result="true" reason=""/>
<test id="12" result="OK" case="Test previous method" expected="true" result="true" reason=""/>
<test id="13" result="OK" case="Test size method" expected="true" result="true" reason=""/>
<test id="14" result="OK" case="Test swap method" expected="true" result="true" reason=""/>
<test id="15" result="OK" case="Test toString method" expected="true" result="true" reason=""/>
<test id="16" result="OK" case="Test update method" expected="true" result="true" reason=""/>
<test id="17" result="OK" case="Test values method" expected="true" result="true" reason=""/>
```

Figura 16. Captura de pantalla d'una col·lecció de tests creada corresponent a l'intent 1 de la llista enllaçada. Font: original.

```
<time value="0:05:11.683819"/>
<test id="1" result="OK" case="Creating empty linked list" expected="0" result="0" reason=""/>
<test id="2" result="OK" case="Adding elements to the list and checking size" expected="3" result="3" reason=""/>
<test id="3" result="OK" case="Removing an element from the list and checking size" expected="2" result="2" reason=""/>
<test id="4" result="OK" case="Setting an element in the list" expected="element1-modified" result="element1-modified" reason=""/>
<test id="5" result="OK" case="Iterating over the list" expected="element1, element2, element3" result="element1, element2, element3" reason=""/>
```

Figura 17. Captura de pantalla d'una col·lecció de tests creada corresponent a l'intent 2 de la llista enllaçada. Font: original.

Com es pot veure, la diferència entre les dues crides és enorme. En la primera, s'han generat fins a 17 tests que verifiquen el comportament del codi generat. Mentre que a la segona, només se n'han generat 5.

També és interessant assenyalar que el nom o descripció que assigna als tests mai és la mateixa. De nou, torna a aparèixer l'alt component d'aleatorietat del LLM.

A més, si s'analitzen els fitxers amb el codi generat, el fitxer que va associat amb els primers tests, conté tots els mètodes sense cos, mentre que el segon, tot i haver creat menys tests per validar el comportament, els seus mètodes sí que tenen cos i, a més, el comportament que té és correcte, tot i no ser el més eficient possible.

```
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 codi generat corresponent a l'intent 1 de la llista enllaçada. Font: 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 codi generat corresponent a l'intent 2 de la llista enllaçada. Font: original.

Per tant, el fet de tenir una col·lecció de tests més gran que una altra, no garanteix que el codi sigui de millor qualitat. Ara bé, cal deixar constància que aquest fet esmentat, ha succeït en una classe que conté bastants mètodes i que durant la generació de codi, hi va haver diversos problemes referents als temps de resposta. De fet, en la Figura 16, s'observa que el temps emprat per generar la classe va ser de més de 5 minuts, ja que va entrar en bucle amb diversos *timeouts*.

D'aquesta manera, és necessari analitzar altres classes amb menys mètodes per acabar de verificar si la quantitat de text que ha de generar influeix en la qualitat del codi generat. Per fer-ho, es mostrarà la col·lecció de tests creada per una cua de dos intents diferents.

```
<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 d'una col·lecció de tests creada corresponent a l'intent 13 de la cua. Font: 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 d'una col·lecció de tests creada corresponent a l'intent 15 de la cua. Font: original.

Si s'analitza el codi, si bé és cert que no tenen exactament el mateix comportament, no existeix la diferència que s'ha produït amb la llista enllaçada.

```java
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 codi generat corresponent a l'intent 13 de la cua. Font: original.

```java
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 codi generat corresponent a l'intent 15 de la cua. Font: original.

4.8. Generació de codi aplicant cadenes de *LangChain*

Pel que fa a aquesta metodologia, s'ha preparat al prototip per poder utilitzar una cadena amb l'objectiu d'enviar més d'un *prompt* al LLM. Concretament, el

primer *prompt* s'ha emprat per enviar un exemple sobre que ha de generar i, en el segon, sol·licitar-li el codi aplicant una de les tres metodologies descrites en els apartats anteriors.

Després de dur a terme diverses proves, s'ha hagut de descartar aquesta metodologia per culpa dels alts temps d'espera que es produïen i, fins i tot, acabaven produint un *timeout* eliminant qualsevol possibilitat de rebre la resposta generada. Addicionalment, aquesta metodologia requereix que existeixi un exemple pel codi que es vol crear. En conseqüència, s'afegeix una dificultat com passa a la metodologia on s'envia una col·lecció de tests ja creats.

Per tant, ha estat una via que no s'ha pogut explorar i no es tindrà en compte en el capítol d'anàlisi. Tot i això, és important destacar que, quan el LLM sigui capaç d'oferir una eficiència acceptable amb aquesta metodologia, serà interessant avaluar-ne els resultats.

4.9. Generació de codi aplicant *active prompting*

Finalment, aquesta ha estat l'última metodologia que s'ha posat a prova. Concretament, com ja s'ha explicat al capítol 3 d'aquesta memòria, es pot aplicar barrejant altres metodologies. Consegüentment, els temps d'espera d'aquesta metodologia depenen directament de l'altra (si no es vol validar el codi amb tests, amb tests propis o creats per ell mateix).

Per tant, tot i no poder-se fer amb una cadena com estava planificat inicialment, s'ha plantejat de manera que es torna a enviar el codi generat amb els comentaris que hagi fet l'usuari que està generant el codi. D'aquesta manera, s'eviten els problemes associats a la poca eficiència de la cadena i es modifica el codi generat inicialment.

Així, la millora del nou codi generat torna a dependre d'un factor, que és la qualitat de la informació que li ha proporcionat la persona que està sol·licitant el codi. De fet, si la informació proporcionada és molt concreta i abasteix tot el que s'ha de modificar, amb una sola crida més al LLM, és capaç de modificar el codi i complir tots els requisits d'aquest.

El procediment que s'ha de seguir per explotar al màxim aquesta metodologia consisteix a avaluar la resposta que s'ha obtingut inicialment. En cas que hi hagi algun aspecte que s'hagi de modificar o replantejar, s'ha d'indicar sempre utilitzant ordres. És a dir, el LLM utilitzat entén i millor les indicacions si aquestes són donades com ordres. I, en cas de ser diverses, que es donin en diferents oracions com si es tractés d'una llista. De fet, aquest tipus de comportament ja s'ha explicat a l'apartat on es parlava del disseny dels *prompts*. Tenint en compte que les indicacions que s'han de donar s'enviaran en un *prompt*, és coherent que també s'hagin d'enviar seguint el mateix format quan s'aplica *active prompting*. Per tant, s'han d'evitar oracions llargues i poc clares i expressions com: *hauria de fer*, *potser estaria millor que* o *crec que*.

Un exemple de missatge que compleix amb els requisits que s'han esmentat, podria ser el següent:

> *Has d'afegir un atribut que permeti comptar el nombre d'elements de la taula. Has d'eliminar tots els comentaris del codi. Modifica el nom del mètode calculateNumber per getSize.*

Com es pot veure, les oracions són curtes, clares i indiquen ordres que ha de seguir.

Finalment, s'ha detectat que, si es duu a terme moltes vegades sobre un mateix codi aquesta metodologia, els resultats que retorna comencen a oblidar part del que s'havia dit inicialment. En conseqüència, no és recomanable iterar molts cops aquesta metodologia sobre un mateix codi. En cas que no retorni el codi esperat, s'obté un millor resultat si es torna a replantejar el *prompt* inicial. De fet, aquesta peculiaritat es dóna perquè el LLM utilitzat no té memòria i, si se supera la limitació de *tokens*, no pot tenir en compte el que se li ha dit inicialment.

Quan es van dur a terme les crides a l'API d'aquest projecte, la limitació de *tokens* estava a 4.096, mentre que en el moment de lliurar aquesta memòria, s'ha augmentat a més de 16.000. Amb aquesta millora, és probable que es puguin dur a terme més iteracions sobre aquesta metodologia, però, un cop superi la nova limitació, començarà a passar el que ha passat durant aquest estudi.

4.10. Errors i problemes detectats durant la generació de codi

Per acabar el capítol, es comentaran diversos errors o comportaments no desitjats que s'han vist generalitzats independentment de la metodologia emprada.

El primer dels errors que es va observar i que s'ha mantingut fins al final del treball, tot i haver aplicat mesures de mitigació, són els *timeouts*. Al principi, a causa de la complexitat que tenia el *prompt* ocorria sempre. Per tant, després de canviar-los, aplicant la tècnica que s'ha descrit en els apartats anteriors, es va aconseguir disminuir el nombre de vegades que apareixien. No obstant això, s'han detectat comportaments que s'han repetit durant tota l'execució de les proves.

Com ja s'ha esmentat en el capítol 3, el prototip disposa d'una manera d'executar-se que li permet generar múltiples vegades una mateixa classe. Inicialment, aquesta opció no es podia utilitzar, ja que s'ha detectat que l'API del LLM utilitzat, detecta si es duen a terme moltes consultes consecutives. Per tant, després de generar un parell de classes, el prototip ja no en generava més perquè sempre retornava un *timeout*.

Amb l'objectiu d'evadir aquesta política de seguretat contra el *spam*, es va haver d'implementar un temps d'espera entre aleatori entre cadascuna de les

crides. Gràcies a això, es va aconseguir mitigar considerablement el problema dels *timeouts*.

Tot i això, hi havia classes que encara continuaven produint aquesta situació. Concretament, es va observar que les classes que tenien més mètodes, tendien més a provocar un error per temps d'espera. Consegüentment, hi ha hagut diverses classes de la biblioteca que no s'han pogut posar a prova per aquest motiu.

Addicionalment, en aquelles classes amb bastants mètodes on sí que es podia executar el prototip, tot i tenir uns temps d'espera considerablement alts, es generava el codi, però, la major part dels cops ho feia amb mètodes sense cos. Concretament, quan ho feia, sempre deixava un comentari dins del mètode tal com es pot veure a la Figura 18.

Fins i tot, hi ha hagut classes on en cap ocasió ha estat capaç de crear mètodes amb cos tot i no haver donat excessius problemes a l'hora de generar el codi. Consegüentment, la falta de cos en aquests mètodes no es deu al fet que tingui un context amb molts *tokens*, sinó que es deu a la falta de comentaris dins del fitxer original des d'on s'obté el context.

```
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 codi font original de la biblioteca de l'assignatura Disseny d'Estructura de Dades del fitxer *DirectedGraphImpl.java*. Font: original.

Com es pot observar a la Figura 24, els mètodes que ha de generar no tenen documentació. Per tant, d'aquest cas se'n pot extreure que el LLM no és capaç de generar el cos del mètode només a partir del nom d'aquest. Inicialment, es pot pensar que el nom del mètode i els paràmetres que rep, poden arribar a ser suficient per donar context al LLM. Amb aquest cas, queda demostrat que no sempre és possible.

De fet, és important destacar que no sempre és possible, deixant espai a situacions on sí que ho és perquè, de fet, hi ha situacions on s'ha observat que és capaç de crear el cos del mètode. Per exemple, la classe *MergeSort* tampoc conté comentaris associats al mètode. Tot i això, hi ha hagut situacions on sí que ha estat capaç de crear el codi.

Consegüentment, la capçalera del mètode esdevé un context insuficient per poder garantir que el codi es podrà generar correctament.

Un altre aspecte rellevant a destacar és que, com que s'han dut a terme múltiples crides a l'API durant tot el semestre, s'ha detectat que en el canvi de versió del dia 3 de maig va provocar que els resultats empitjoressin dràsticament amb el *prompt* que s'estava utilitzant anteriorment. De fet, el canvi va tal, que va passar de generar codi correcte en un nombre reduït de vegades a directament generar respostes com la següent.

Figura 25. Resposta obtinguda en executar el prototip just després d'un canvi de versió amb el *prompt* funcional anterior.

En conseqüència, s'ha detectat que es poden produir diferents resultats utilitzant exactament el mateix *prompt* en diferents versions. A més, el canvi és molt dràstic, ja que es va passar de generar sempre codi (tot i que moltes vegades el generava amb mètodes sense cos) a generar respostes que directament no contenien codi. De fet, el test *(code here)* és una part del *prompt* que es va enviar.

Continuant amb els errors que s'han pogut detectar, s'ha observat que hi ha diverses franges horàries on es generaven més *timeouts* que altres. Per tant, sembla evident que, quan el model està més monopolitzat, és més difícil poder generar el codi i, en general, obtenir respostes amb l'API. Així, es va haver d'anar fent proves a diverses franges horàries per poder trobar la que millor rendiment oferia.

Després de diversos dies, es va poder concloure que la millor franja horària per llençar les crides a l'API és entre les 2 AM i les 6 AM. És a dir, la franja horària que coincideix quan és de nit a Europa i quan la jornada laboral, en principi, hauria d'haver acabat a Amèrica. A més, corroborant aquest fet, també s'ha detectat que durant els dies que cauen en cap de setmana, es redueix el nombre de *timeouts*.

Ara bé, aquesta conclusió és una arma de doble fil, ja que si les altres persones usuàries del LLM detecten aquesta situació, podran automatitzar les crides que

hagin de fer per fer-les durant les hores esmentades. Així, tot i que aquesta afirmació s'ha pogut fer durant els primers mesos de vida de la versió del LLM utilitzat, en el futur pot canviar dràsticament.

El següent punt a tenir en compte, tot i no ser un error com a tal, és un comportament estrany i que cal ser reportat. Concretament, hi ha hagut mètodes on el LLM ha afegit etiquetes com *@Override*, tot i no necessitar-ho. A més, en algunes ocasions, ha entès que calia heretar classes que no eren necessàries. Per exemple, heretar la classe *Object* en una classe de Java.

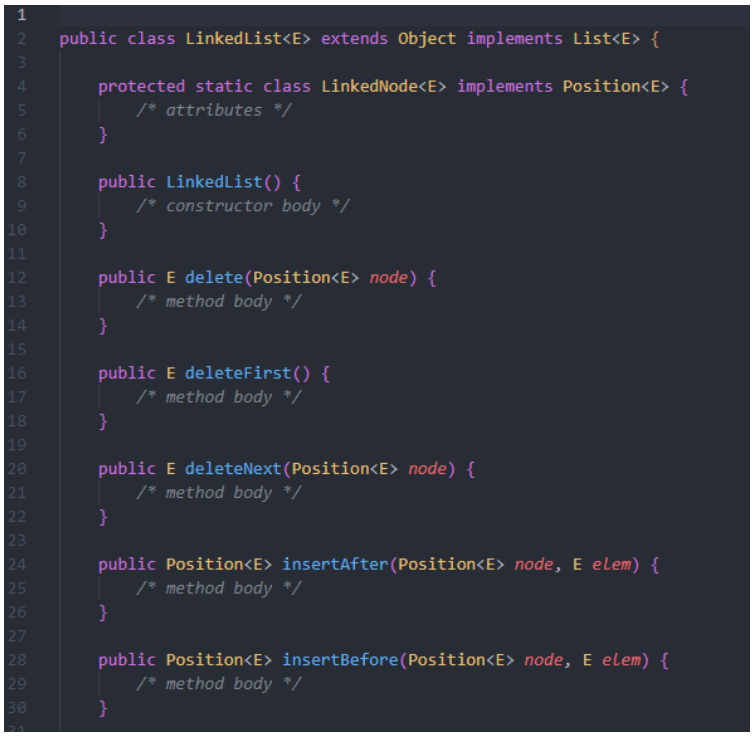
Pel que fa als tests generats, s'han trobat diverses situacions on aquests mostren una realitat que és impossible que sigui certa. Concretament, existeixen tests que són falsos negatius, on s'indica que un test ha fallat, però realment ho ha fet per falta de context. És a dir, és un error que no hauria de ser rellevant quan es tracti amb el codi generat. En canvi, hi ha situacions on es mostra que tots els tests han passat correctament, però, en canvi, el cos dels mètodes està buit. Per tant, els tests estan mentint sobre el correcte funcionament del codi.

Aquest fet s'ha pogut detectar amb la classe *LinkedList* a l'intent número 9.

```
1  <time value="0:04:42.783682"/>
2  <test id="1" result="OK" case="Creating an instance of an empty linked list" expected="true" result="(new
   LinkedList<Integer>()).isEmpty()" reason=""/>
3  <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=""/>
4  <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=""/>
5  <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=""/>
6  <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 on s'indica que tots els tests generats passen correctament.
Font: original.

En canvi, si es revisa el codi generat, es confirma que és completament impossible que els tests s'hagin superat correctament.



```
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 de l'inici del codi generat de la classe *LinkedList* sense cos als mètodes. Font: original.

En conseqüència, pot haver-hi situacions on els tests tinguin un resultat que no és possible. Aquests casos s'han observat especialment en aquelles classes que envien un context molt gran al LLM.

Finalment, un altre problema que s'ha detectat, que ja s'ha introduït anteriorment, és que com més llarg és un context que s'envia, més probabilitats hi ha que no segueixi alguna de les regles dictades al *prompt*. Com que el prototip es basa a oferir una resposta en forma XML, quan aquesta no manté aquest format, mostra un error. I, concretament, aquest error només apareixia quan la quantitat de mètodes que s'havien de generar i el context que s'envia era considerablement més llarg que altres.

```
Traceback (most recent call last):
  File "D:\git\UOC\codegenerator\__main__.py", line 124, in <module>
    CodeWithCustomTests(model, text, tests, folderName)
  File "D:\git\UOC\codegenerator\__main__.py", line 45, in CodeWithCustomTests
    generator.Generate()
  File "D:\git\UOC\codegenerator\src\GenerateCodeWithCustomTests.py", line 22, in Generate
    super().SaveAnswer()
  File "D:\git\UOC\codegenerator\src\CodeGenerator.py", line 144, in SaveAnswer
    raise Exception("[ERROR] Cannot find <tests> tag!!")
Exception: [ERROR] Cannot find <tests> tag!!
```

Figura 28. Error que il·lustra la situació que es donava quan la resposta no seguia el format sol·licitat.

5. Anàlisi comparatiu

5.1. Anàlisi per variables i metodologia

L'anàlisi que s'ha dut a terme en aquest capítol no conté totes les gràfiques obtingudes. Aquestes han estat tractes i s'han seleccionat només aquelles que són més significatives per analitzar i comparar els resultats. No obstant això, si el lector d'aquest document desitja consultar-les, es pot accedir al repositori de gràfiques mitjançant l'Annex VI. Addicionalment, si també es desitgen consultar tots els resultats aconseguits directament de les crides al LLM, es poden consultar dins de la carpeta *data* del repositori del codi a l'Annex IV.

Per tant, en aquest capítol es durà a terme una anàlisi entre les diferents metodologies que s'han aplicat respecte a diverses variables.

En primer lloc, es farà una anàlisi sobre la qualitat del codi generat, és a dir, el que s'ha anomenat com a precisió. Només es tractaran les tres metodologies que s'han automatitzat, és a dir, la generació de codi sense validació amb tests, la que valida el codi amb tests proporcionats en el mateix *prompt* i, finalment, la que valida el codi generat amb una col·lecció de tests pròpia.

En segon lloc, es compararà quin és el grau de millorar cada cop que s'itera el codi. Per simular una situació real, s'entendrà com a iteració cada cop que un usuari interactua amb el LLM proporcionant informació respecte al codi prèviament generat.

Finalment, cal dur a terme un estudi sobre l'eficiència temporal que té cadascuna d'aquestes metodologies, ja que és important analitzar si són viables.

5.1.1. Precisió

En aquest apartat, es farà una anàlisi comparativa entre les tres metodologies emprades segons la qualitat del seu codi. Com que no es poden presentar totes les gràfiques de totes les classes generades, s'han seleccionat aquelles que es consideren més rellevants per constatar certs aspectes que ja s'han introduït anteriorment. És a dir, s'han seleccionat les gràfiques més significatives i representatives.

A més, tal com s'ha dit en el capítol anterior, hi ha hagut classes que directament no han generat cap mètode amb cos. En conseqüència, no es pot dur a terme una anàlisi comparativa sobre aquestes classes, perquè no han generat un codi vàlid en cap cas.

Abans, però, s'ha de definir en funció de quines variables es determinarà la precisió del codi. Concretament, s'estableixen 5 categories possibles.

- Sense codi: representa aquells casos on s'han generat els mètodes, però aquests estan sense cos.
- Malament: aquesta categoria representa els intents on el codi generat, tot i ser sintàcticament correcte, no ofereix el comportament desitjat.
- No eficient: representa aquells casos on el codi permet oferir un funcionament correcte, però de manera no eficient.
- Correcte: presenta aquells casos on el codi generat ofereix el comportament sol·licitat i és eficient, però no té en compte el context de la classe i defineix alguns components que no són necessaris, ja que vénen donats per les classes pare.
- Perfecte: són aquells codis que generen un codi que compleix amb les condicions per ser considerat correcte i, a més, té en compte el context de la classe i utilitza mètodes o classes que venen heretades.

La primera de les classes seleccionades, amb la qual s'ha estat treballant tot el semestre és l'anomenada *QueueArrayImpl*. Aquesta classe és especialment interessant perquè permet categoritzar a la perfecció els codis generats en funció de les categories anteriors. Concretament, la versió més eficient d'aquesta classe és la que utilitza el vector on emmagatzema els elements com si fos un anell. Per tant, permet quantificar quin és el percentatge de classes que s'han pogut generar tenint en compte aquesta optimització.

Addicionalment, aquesta classe no conté un gran nombre de mètodes a generar. Així, s'utilitzarà com a classe representativa d'aquelles que tenen un context més curt.

Finalment, amb l'objectiu de poder analitzar les diverses metodologies entre les diferents versions que hi ha hagut durant els últims mesos, es mostraran les gràfiques associades a la classe mencionada per la versió del dia 3 de maig i, seguidament, per la versió del dia 24 de maig.

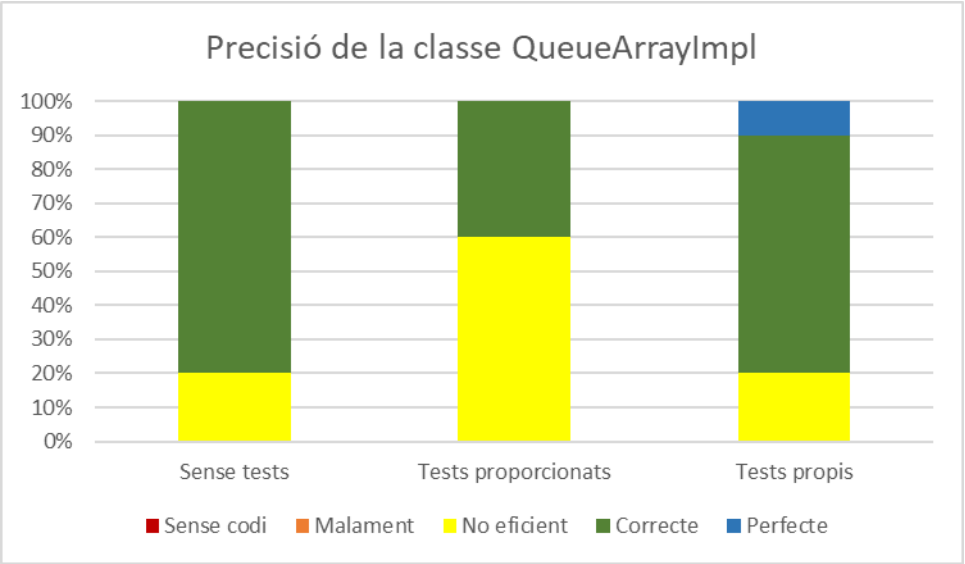


Figura 29. Gràfica que representa la precisió del codi generat de la classe *QueueArrayImpl* amb la versió del dia 3 de maig. Font: original.

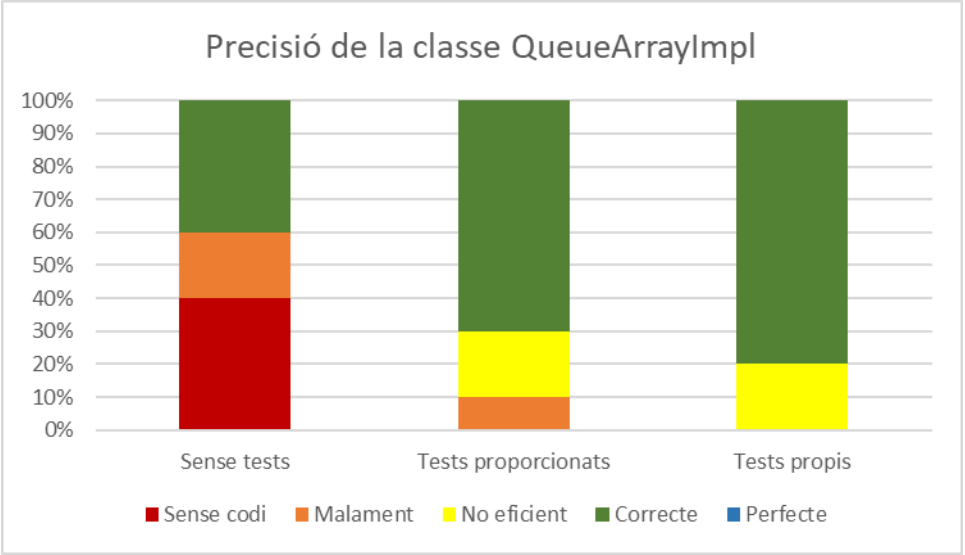


Figura 30. Gràfica que representa la precisió del codi generat de la classe *QueueArrayImpl* amb la versió del dia 24 de maig. Font: original.

Com es pot apreciar a primera vista, ja hi ha hagut un canvi considerable entre les dues versions amb les quals s'ha executat el codi utilitzant exactament el mateix *prompt* per les dues. Si bé és cert que el nombre d'intents no és considerablement alt, 10 vegades cada metodologia per versió (un total de 60 fitxers analitzats), ja es poden veure diferències notables.

Per exemple, amb la versió del dia 3, els resultats obtinguts quan no es validava el codi amb tests no generaven mai mètodes sense cos. Amb el canvi de versió, s'ha observat que ha empitjorat notablement aquesta metodologia.

Tot i això, sí que s'observen patrons que es mantenen entre les dues gràfiques, és a dir, hi ha similituds pel que fa a les metodologies emprades entre diverses versions. Concretament, en ambdues gràfiques es manté que la que ofereix millors resultats és la que genera els tests per ella mateixa. Aquest fet s'ha estat comentat durant tota la memòria i amb aquestes gràfiques es pot constatar que és cert.

Consegüentment, la metodologia que no utilitza tests com a validació, no sembla una bona elecció tenint en compte que els resultats són molt inestables. A més, veient els resultats, tenint en compte que la metodologia on es proporciona una col·lecció de tests requereix que es desenvolupin manualment, també es conclou que la millor metodologia, segons les gràfiques, és la que genera els tests per ella mateixa, ja que permet obtenir millors resultats sense haver de generar els tests prèviament.

Ara bé, és necessari dur a terme un estudi sobre quin és el comportament que han tingut els tests. Concretament, s'analitza si els tests s'han superat correctament, si n'hi ha algun que està marcat com a fallat, però no és un error rellevant o si ha fallat perquè realment el codi és incorrecte. Així, les següents gràfiques mostren els resultats dels tests per les dues metodologies que els usen.

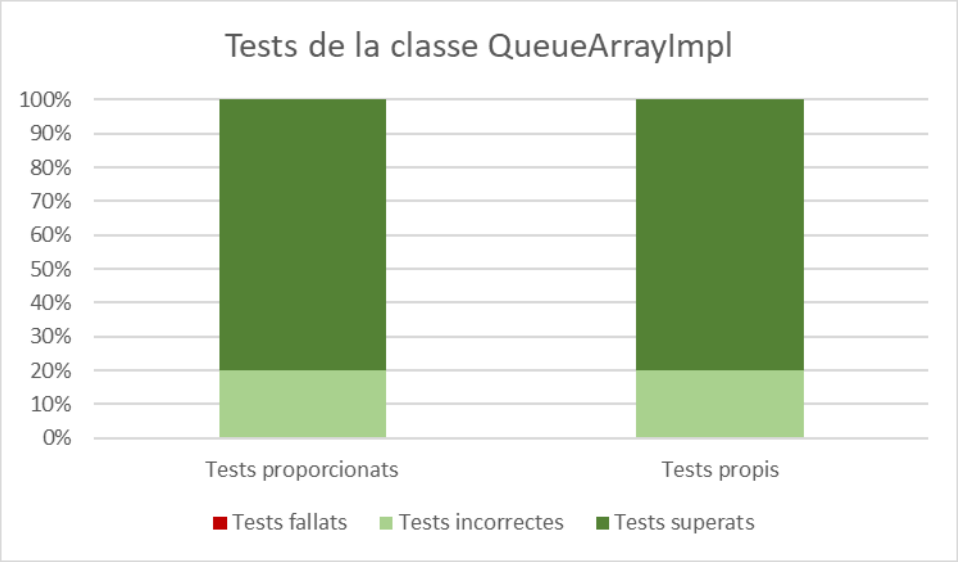


Figura 31. Gràfica que representa la correctesa dels tests de la classe *QueueArrayImpl* amb la versió del dia 3 de maig. Font: original.

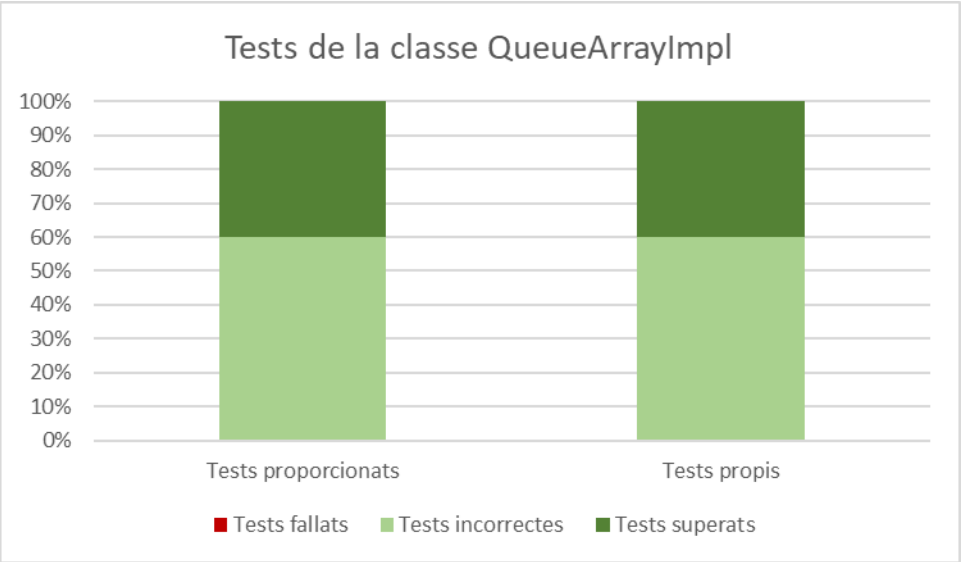


Figura 32. Gràfica que representa la correctesa dels tests de la classe *QueueArrayImpl* amb la versió del dia 24 de maig. Font: original.

Continuant amb la tendència de les gràfiques anteriors, s'observa com hi ha un empitjorament entre les dues versions pel que fa a la qualitat dels tests, ja que han augmentat els casos on fallen tests, però realment no és correcte. Tenint en compte que s'han analitzat 10 casos per a cada metodologia, aquest nombre per tenir un cert biaix perquè la població usada no és molt elevada. El que sí que es pot afirmar és que la proporció entre les dues metodologies continua sent la mateixa.

En conseqüència, pel que fa a aquelles classes que no tenen un context molt gran a enviar, es conclou que la millor metodologia que es pot aplicar és la que genera els tests per ella mateixa.

En canvi, si s'analitzen aquelles classes que tenen un context molt més gran, els resultats canvien dràsticament. Per analitzar aquests resultats, s'utilitzarà la classe *LinkedList* com a classe representativa, ja que conté un gran nombre de mètodes a desenvolupar.

A continuació, sense abandonar el mateix model de gràfiques que amb la classe anterior, la gràfica obtinguda és la següent:

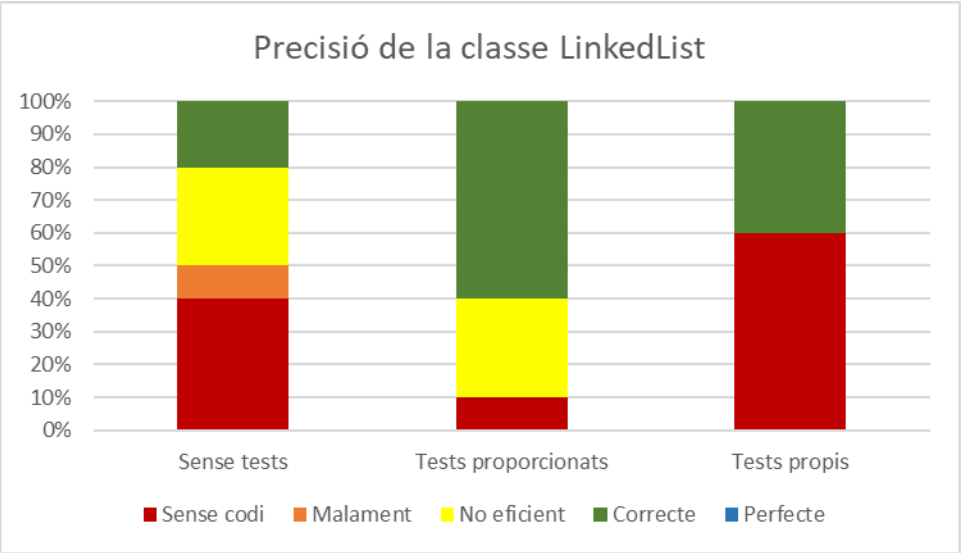


Figura 33. Gràfica que representa la correctesa dels tests de la classe *LinkedList* amb la versió del dia 24 de maig. Font: original.

Com es pot apreciar, els resultats empitjoren considerablement. En línies generals, destaca especialment que han augmentat els casos on no hi ha codi. Per tant, es confirma el que s'ha anat dient al llarg de tota la memòria i s'arriba a la conclusió que, quan el context és considerablement llarg, els resultats empitjoren.

Ara bé, en els 10 casos analitzats, crida l'atenció que la metodologia que usa tests propis, té molts casos on no hi ha cos als mètodes, però, quan aquest hi és, la solució que proposa és correcte. Per tant, aquesta metodologia permet confirmat que, efectivament, els resultats no són els esperats per culpa de la gran quantitat de *tokens* que s'estan enviant, fet que provoca que no es generi el codi dins dels mètodes. Si el problema fos la metodologia, en els casos on s'hagués generat codi, es mostrarien escenaris on el codi generat no és eficient o, directament, és erroni.

Així, doncs, aquestes són les gràfiques que millor representen els dos casos citats: quan el context no és extremadament llarg o quan sí que ho és.

5.1.2. Escalabilitat de precisió per iteració

Tal com s'ha comentat anteriorment a l'apartat 4.9 d'aquesta memòria, la millora de la precisió del codi generat depèn totalment de la informació que l'usuari doni en aquell moment en funció del resultat obtingut. Consegüentment, esdevé impossible oferir gràfiques amb dades útils, ja que depenen de l'habilitat que tingui la persona que interactua amb el LLM.

Addicionalment, després de diverses proves s'ha pogut observar que la millora és tal que fins i tot, partint d'un codi amb els mètodes sense cos, es pot arribar a una situació on el codi sigui perfecte. Ara bé, com pitjor estigui el codi generat en una primera instància, més indicacions cal fer.

Per acabar, tenint en compte que el LLM no té una capacitat d'anàlisi lògica, si l'usuari és capaç de proporcionar-li aquest punt de vista, el codi generat aplica realment els canvis sol·licitats correctament, fins i tot aquells que tenen a veure amb plantejaments lògics. Com a resultat, es torna a un dels punts esmentats a la introducció on s'afirmava que la generació de codi utilitzant LLM no pot reemplaçar en cap cas a la persona que programa, ja que aquesta ha de tenir un coneixement per poder donar-li indicacions al LLM. En poques paraules, el LLM permet agilitzar considerablement el desenvolupament de programari, però no substituir a la figura del programador.

A tall d'exemple, reutilitzant el cas de l'anell que ha d'utilitzar una cua per estar desenvolupada eficientment, es mostra el resultat d'una primera crida on no tracta la cua de manera correcta i, amb la interacció d'un desenvolupador, s'aconsegueix aplicar correctament un plantejament lògic.

```
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 mètode per afegir un element a la cua de manera ineficient. Font: original.

Com que el plantejament que ha utilitzat no és correcte se li indica que l'ha de canviar amb la següent frase.

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

Un cop enviada la crida al LLM, el codi resultat esdevé correcte i eficient.

```
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 mètode per afegir un element a una cua de manera eficient. Font: original.

En resum, doncs, queda demostrat que amb una sola iteració, el resultat pot ser el desitjat.

5.1.3. Eficiència temporal

Un dels aspectes importants a analitzar és la viabilitat temporal, és a dir, estudiar si és viable la generació de codi amb aquestes metodologies i, a més, fer una comparació entre elles.

Per analitzar l'eficiència temporal, doncs, cal tenir en compte que durant les crides a l'API, hi ha hagut diversos problemes referents als temps d'espera que ja s'han comentat anteriorment. Per tant, a les següents gràfiques s'observaran diversos *outliers* que no s'han de tenir en compte a l'hora de valorar una metodologia, tot i que se'n deixarà constància.

Una de les classes que millor representa la proporció de temps entre metodologies és la classe *PriorityQueue*.

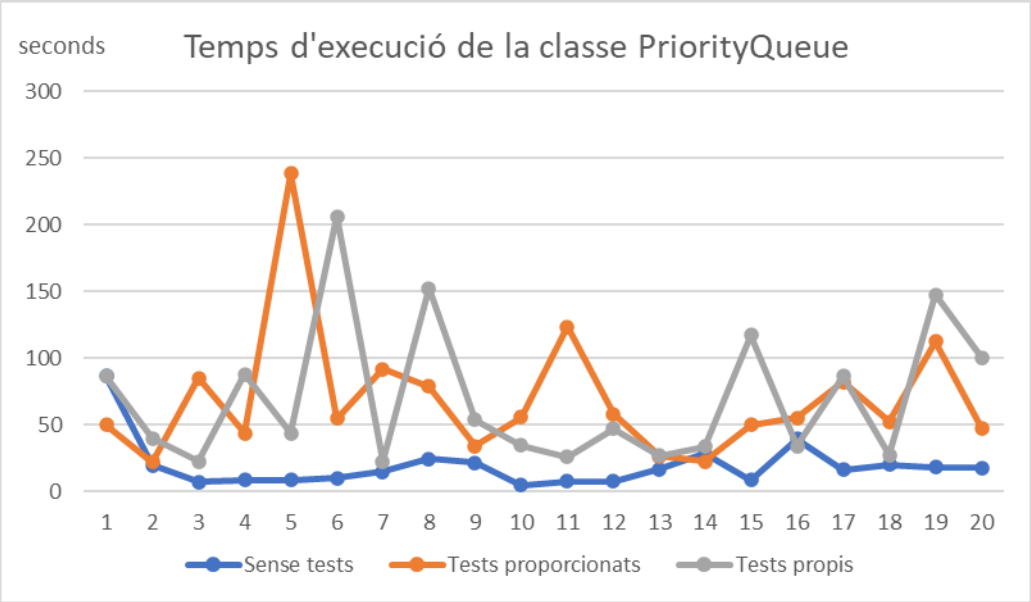


Figura 36. Gràfica que representa el temps usat per generar el codi de la classe *PriorityQueue* segons la metodologia.

A primera vista ja es pot veure que la metodologia que menys temps necessita i la que més estable és, és la que no valida el codi amb tests. De fet, és normal que sigui així, ja que és la metodologia menys context s'envia i, en conseqüència, és la més eficient temporalment. Ara bé, com ja s'ha vist abans, aquesta és la que pitjors resultats ofereix.

Per tant, analitzant les altres dues metodologies, el primer aspecte que crida l'atenció és poca regularitat que hi ha. Això és degut als *timeouts* que s'han anat produint durant la generació de codi, que feia que augmentés considerablement el temps d'espera. Ara bé, no sempre es produïen, en conseqüència, la gràfica generada és totalment irregular.

Tot i això, quan no es produeixen *timeouts*, el temps d'espera és lleugerament superior al que es produeix amb la metodologia que no utilitza tests. Doncs, com que ofereixen millors resultats i la diferència, eliminant els *outliers*, no és tan gran, esdevenen metodologies temporalment vàlides.

S'ha de tenir en compte que, per considerar una metodologia no vàlida, s'hauria de parlar en l'ordre de bastants minuts, ja que s'ha de tenir en compte que actualment el codi el desenvolupa una persona des de zero. En canvi, si s'utilitza aquesta eina, es redueix considerablement el temps total de desenvolupament perquè es pressuposa que l'usuari acabarà modificant el codi generat pel LLM.

Pel que fa a les altres classes, n'hi ha hagut algunes que han estat molt més estables que aquesta que s'acaba de comentar. No obstant això, aquestes gràfiques no mostren una realitat, ja que el codi generat amb aquestes classes no contenia en cap cas mètodes amb cos definit. Consegüentment, no té sentit comentar-les detalladament perquè el seu valor és nul.

El que sí que mereix ser comentat és la gràfica de la classe *Stack*.

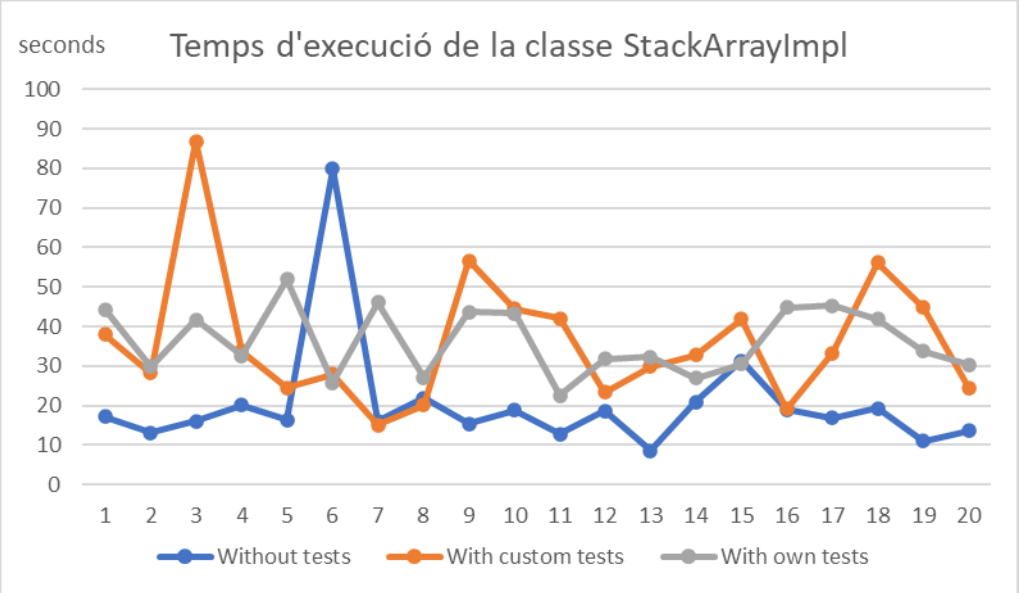


Figura 37. Gràfica que representa el temps usat per generar el codi de la classe *StackArrayImpl* segons la metodologia.

La part interessant d'aquesta gràfica, a diferència de l'anterior és que la quantitat de temps utilitzat és més semblant entre metodologies. Això és degut al fet que la classe generada té molts menys mètodes a desenvolupar. Per tant, d'aquesta gràfica s'arriba a conclusió que, com menys mètodes s'hagin de desenvolupar, els temps d'execució seran més semblants entre metodologies. Tot i això, la metodologia que no genera tests, en línies generals, continua sent la més eficient temporalment.

5.2. Avantatges i desavantatges de cada metodologia

Pel que fa als avantatges i els desavantatges de cadascuna de les metodologies s'han comentat de manera indirecta durant la memòria. Tot i això, és important fer-ne un resum i tenir en compte aquests aspectes a l'hora de triar-ne una o una altra.

Sense tests

Pel que fa a aquesta metodologia, s'ha observat que és la que ofereix una millor eficiència temporal. Addicionalment, també és una metodologia que s'aplica sense necessitat d'haver de generar cap mena de material manualment, ja que no s'envia res més que el llenguatge natural definit per l'usuari sobre que vol generar.

Ara bé, aquesta també és la que ofereix uns pitjors resultats davant de les altres dues. On, a més, destaca que en múltiples ocasions el codi generat contenia mètodes sense cos. A més, els resultats tampoc milloraven quan el context enviat era més gran.

Consegüentment, no és una metodologia que sigui recomanable a l'hora d'usar-la en un entorn professional.

Amb tests proporcionats

En canvi, aquesta metodologia presenta moltes diferències respecte de la que s'acaba de comentar. Concretament, una de les millores és que ofereix uns resultats força bons, però no suficients per poder afirmar que és una metodologia a seguir.

A més, és la metodologia que millor ha tolerat els canvis que s'han apreciat entre el canvi de versió del LLM.

Un dels altres inconvenients que presenta aquest cas és que requereix que existeixi una col·lecció de tests prèviament desenvolupament manualment. Per tant, al temps d'execució quan es genera el codi requerit, se li ha de sumar el temps que es tarda a desenvolupar els tests.

Així, doncs, aquesta metodologia, tot i no tenir uns resultats molt dolents, es conclou que tampoc és la més adequada, ja que té un cost temporal considerablement alt.

Amb tests propis

Finalment, l'última metodologia presenta múltiples avantatges. El primer d'ells, que és el més important, és que presenta els millors resultats possibles. Tot i que a mesura que augmenta el context que s'envia al LLM, els resultats comencen a empitjorar.

Per altra banda, també ofereix avantatges perquè no necessita que es creï cap col·lecció de tests manualment. La mateixa metodologia s'encarrega de generar-los. Com a resultat, no s'ha de sumar cap altre temps com sí que passava amb l'anterior metodologia.

Com a efectius negatius, s'ha de destacar que ha empitjorat els resultats respecte a l'última versió amb la qual es van llençar les proves. Tot i això, no és la metodologia que pitjors resultats ofereix.

Finalment, també cal dir que és la metodologia amb el pitjor cost temporal de totes. Tal com s'ha comentat, com que l'ordre de temps del que s'està parlant, excloent els *outliers*, és de segons, es considera que la metodologia és viable temporalment.

Per tant, concloent aquest capítol, segons totes les anàlisis comparatives que s'han dut a terme, s'arriba a la conclusió que la millor metodologia, tot i que no és perfecte, que s'hauria d'utilitzar segons els estudis fets, és la tercera, on el mateix LLM és qui valida el codi amb una col·lecció de tests pròpia.

6. Desplegament del prototip dins d'un IDE

Un cop analitzades les diferents metodologies de generació de codi, es pot posar en pràctica dins d'un IDE per tal que aquest pugui afegir el codi generat directament dins del fitxer de text on s'estigui treballant.

Per fer-ho, s'ha partit de la premissa que és important mantenir certa privadesa sobre com funciona internament el prototip desenvolupat. És a dir, l'usuari final no necessita tenir coneixement sobre quin és el *prompt* que s'està utilitzant per generar el codi ni com es generen les crides a l'API. Per tant, s'ha optat per ocultar aquesta informació afegint un servidor sobre el qual s'executi el prototip. Justament per aquest motiu s'ha mencionat anteriorment que el prototip ha de tenir la capacitat de ser multiplataforma, ja que d'aquesta manera és molt més fàcil fer-lo entrar en producció. A més, si el prototip s'executa en un servidor, qualsevol IDE podrà sol·licitar-li codi de manera senzilla perquè només s'haurà d'implementar el codi referent al client, és a dir, a l'IDE.

Així, el primer que s'ha fet és donat d'alta un servidor a AWS (Amazon Web Services) i s'ha creat un entorn amb Flask per poder habilitar una adreça que accepti consultes de tipus POST. On es requereixi un paràmetre que indiqui el que vol generar el programador. D'aquesta manera, com ja s'ha avançat, el servidor serà qui farà la consulta a l'API del LLM aplicant un *prompt* concret i retornarà la resposta a l'IDE que ha fet la consulta.

El codi que s'executa en el servidor és una versió reduïda del prototip que es pot trobar a l'Annex IV, ja que només requereix la metodologia on s'han obtingut millors resultats. A més, aquesta també permet generar tests de manera automatitzada en funció del codi generat i del *prompt* rebut.

Consegüentment, amb el prototip funcionant dins d'un servidor de manera generalitzada, ja es pot desenvolupar el codi per a qualsevol IDE que ho permeti per fer-li consultes.

6.1. Visual Studio Code

Visual Studio Code ofereix la possibilitat d'afegir extensions dins d'ell, siguin pròpies o de tercers. Per tant, és l'entorn ideal per posar a prova el prototip en un entorn professional. Concretament, per poder-ho s'ha creat una extensió amb JavaScript que només es permet executar de manera local, és a dir, no s'ha publicat l'extensió, ja que aquesta està en una fase experimental. Tot i això, amb l'objectiu de permetre entendre millor el funcionament d'aquesta, s'ha habilitat un repositori públic on es pot consultar el codi d'aquesta. El repositori es pot trobar dins de l'Annex VII.

L'objectiu d'aquest treball no és mostrar tot el procés pel qual s'ha passat per poder crear l'extensió, però a tall d'exemple es mostrarà quin és el funcionament d'aquest.

Concretament, s'ha afegit una nova opció dins del *context menu* que actua com a disparador de l'extensió un cop es prem al botó dret del ratolí si s'ha seleccionat text prèviament.

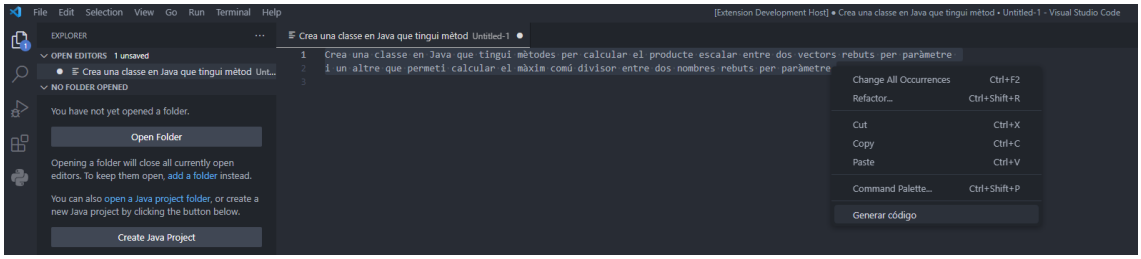


Figura 38. Captura de pantalla on apareix l'opció per generar el codi sol·licitat amb el prototip desenvolupat. Font: original.

Un cop s'activa, es fa una crida al servidor on s'ha desplegat el prototip per poder iniciar tot el procés. D'aquesta manera, el servidor sol·licita al LLM el codi utilitzant el *prompt* amb el qual s'ha treballat durant tot l'estudi i retorna la resposta en format XML.

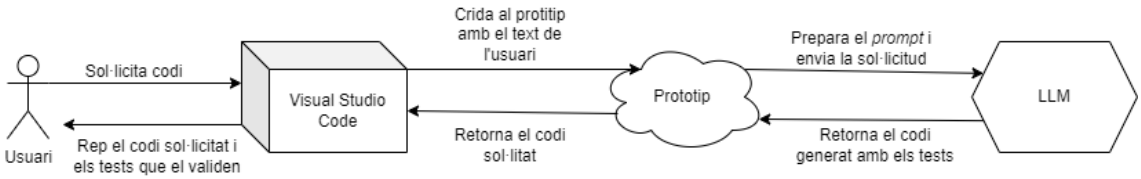


Figura 39. Model conceptual del sistema desenvolupat. Font: original.

Tot seguit, l'IDE afegeix el codi generat dins del fitxer de text des d'on s'ha sol·licitat i mostra pel canal de sortida els tests generats amb el resultat de l'execució.

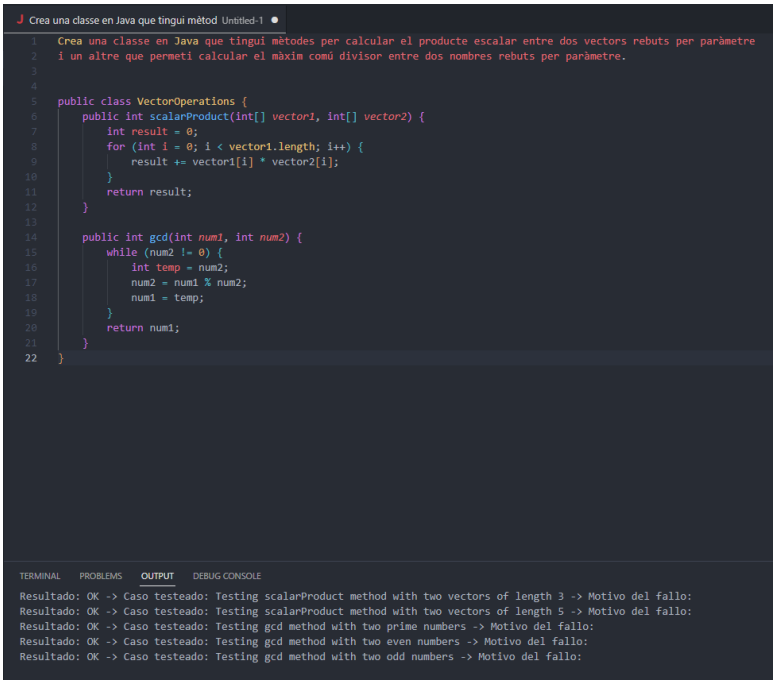


Figura 40. Captura de pantalla amb el codi afegit correctament dins del fitxer de text i el conjunt de tests amb els seus corresponents resultats. Font: original.

Finalment, cal destacar que el temps d'espera depèn completament del codi que es vulgui generar, però pel cas plantejat a la Figura 38 el temps d'espera no acostuma a ser superior a 10 segons. Consegüentment, es considera que el temps de resposta és considerablement bo, tenint en compte el temps que es tardaria a desenvolupar el codi des de zero.

S'ha de considerar que el prototip desenvolupat és una primera versió i que està en una fase experimental. Consegüentment, és bastant probable que sorgeixin problemes inesperats, especialment quan el text introduït per l'usuari és molt llarg. Aquests, haurien de ser detectats i solucionats en futures versions del prototip.

6.1.1. Retroalimentació amb ChatGPT

Pel que fa a la capacitat que ofereix el prototip de comunicar-se amb el LLM utilitzat, s'ha de dir que la comunicació es produeix de dues maneres.

La primera es produeix en la mateixa generació de codi, quan el LLM proporciona una col·lecció de tests creada per ell mateix i ofereix informació sobre el seu mateix codi. D'aquesta manera, la persona que ha sol·licitat el codi pot tenir en compte les consideracions que s'han fet i modificar, o no, el codi generat.

La segona té a veure amb *active prompting*. Si bé és cert que actualment el prototip no funciona usant una única cadena amb múltiples respostes que es van afegint, l'usuari té la capacitat d'escriure quines modificacions s'han de fer i, amb la selecció d'aquest nou text i el codi generat, pot tornar a sol·licitar al prototip que modifiqui el codi generat.

Consegüentment, tot i no oferir mecanismes més eficients, com el cas d'una única cadena, l'usuari té la capacitat de comunicar-se directament amb el LLM mitjançant els tests que aquest genera.

7. Conclusions i treballs futurs

Com s'ha pogut detallar al llarg de tot el treball, l'aplicació de certes metodologies en la generació de codi fa que el resultat pugui millorar considerablement. Ara bé, donat el context que hi ha hagut durant aquest estudi, no es donen les condicions per poder oferir la propietat de ser un sistema determinista. És a dir, els resultats no sempre són els mateixos i tampoc són sempre els esperats.

Com a conseqüència, aquest model es pot utilitzar com a eina que millora considerablement l'eficiència temporal d'una persona que desenvolupa codi. En cap cas, es pot considerar avui en dia que un LLM pot substituir un programador. Per tant, també es pot afirmar que no es pot delegar la generació de codi de manera no supervisada donada la situació actual.

Ara bé, això no implica que els avenços que s'han produït en aquest camp siguin molt interessants. Tal com s'ha vist en el capítol 2 d'aquest treball i s'ha corroborat amb els resultats obtinguts, l'ús dels LLM en el futur dins de la generació de codi ha d'anar en augment. No només ho indiquen l'enorme quantitat de *papers* que s'esmercen en aquest camp, sinó que també ho diuen els resultats que s'han aconseguit en aquest mateix treball.

L'aplicació de la metodologia TDD enfocada a validar el codi generat per un LLM mostra signes inequívocs d'una millora considerable respecte al que s'obté sense ella. Tot i això, l'aplicació d'aquesta metodologia s'ha d'enfocar d'una manera diferent. En el moment de la realització d'aquest projecte, no hi havia certes eines disponibles que ara sí que hi són. Per exemple, no es podien utilitzar *plugins* vinculats a un LLM. I, de fet, aquesta possibilitat, obre el camí que segons diversos estudis citats en aquest mateix document i la conclusió a la qual s'ha arribat amb aquest treball és la necessitat de poder utilitzar diverses eines controlades per un LLM.

Aquesta necessitat s'ha vist reflectida en aquest mateix document quan s'ha afirmat que s'han trobat classes generades amb mètodes sense cos on el LLM afirmava que els tests passaven tots correctament. Com a resultat, si es disposés d'un compilador al qual el LLM tingués accés, podria generar el codi dels tests per poder-los executar en un compilador real.

Ja s'ha avançat en els primers capítols d'aquest treball que un LLM no té la capacitat de compilar i executar el codi. La validació del codi utilitzant tests la fa a partir del que considera que passarà segons el que ha llegit i el que ha redactat.

En aquest sentit, referenciat de nou al model EvalPlus del capítol 2, seria molt interessant modularitzar la generació de codi amb diverses crides al LLM emprant diferents models: un per generar uns tests de qualitat, que podria ser un model com EvalPlus i, l'altra que permeti generar el codi donats els tests que s'han generat i, finalment, poder compilar i executar el codi generat amb els tests. Només d'aquesta manera es podria arribar a una situació on es pugui

afirmar que el sistema és determinista. Fet que farà millorar la confiança de la comunitat cap a aquestes eines.

Ara bé, tot i poder oferir uns resultats molt millors, s'han de considerar diversos aspectes que poden ser un problema que provoqui que el model proposat no sigui possible. Per exemple, s'hauria d'analitzar el cost temporal que tindria un model així i veure si realment és capaç de poder fer sol·licituds com la que s'acaba de suggerir. A més, continuant amb la línia del que s'ha dit anteriorment, els LLM manquen d'una part lògica que podria millorar els resultats obtinguts. Així, de manera indirecta, com que s'arribaria a una situació on els resultats són correctes d'una manera més ràpida, milloraria la complexitat temporal del model.

És a dir, els LLM per ells mateixos són capaços de proporcionar una ajuda considerable. Però, si l'objectiu que es persegueix és dotar a un model de més independència, s'han de proporcionar les eines que s'han esmentat, construint d'aquesta manera un model molt més complex i sofisticat.

Addicionalment, pel que fa al prototip i l'extensió desenvolupats, s'haurien de provar en altres entorns, llenguatges i generat codi del qual el LLM potser no té tant coneixement. En aquest estudi s'han sol·licitat classes en Java d'ADT que són bastant coneguts. Per tant, s'hauria d'anar més enllà i sol·licitar-li codi del qual no té tant coneixement.

Un exemple, podria provar l'extensió creada dins d'una aplicació professional que requereixi codi no molt comú. Per exemple, fent una metàfora amb la universitat, es podria sol·licitar que generés una classe que representi un estudiant de la UOC. En sol·licitar-li això, automàticament requereix més context, ja que el LLM no pot saber que un usuari ha d'estar relacionat amb aules del campus o que hagi de tenir un tutor o tutora assignat.

Consegüentment, s'introdueix un dels altres aspectes claus en el futur: el nombre de *tokens* que es poden enviar com a context. Durant l'estudi el nombre màxim de *tokens* era 4.096. Actualment, l'han augmentat fins a més de 16.000. I, de fet, en el futur, la tendència és que aquest continuï augmentat. En conseqüència, existeix un alt grau de dependència sobre aquest nombre.

Si progressivament el nombre va augmentant, pot arribar un punt on el LLM es pugui introduir dins de qualsevol projecta, metafòricament parlant. És a dir, es pot donar una situació on el LLM tingui coneixement de tot el codi d'una aplicació o servei. Depenent de com sigui de gran l'aplicació, menys *tokens* seran necessaris. Però, si en un futur és viable donar tot el context d'una aplicació, els resultats millorarien brutalment i, en conseqüència, s'estaria arribant a una situació on el LLM podria arribar a ser capaç de generar codi segons les necessitats que introdueixi un programador, per molt gran que aquesta sigui.

En qualsevol cas, aquesta situació esmentada és ciència-ficció actualment i queda considerablement lluny de l'abast d'aquest treball, però sense cap dubte és un escenari que ben segur voldrà ser explorat en un futur. Depèn de com de

ràpid s’avanci computacionalment en aquest camp. La cursa ja fa anys que ha començat.

“Desafiem les fronteres de la tecnologia i la ciència amb curiositat i valentia cap a una singularitat que ens redefineixi.”

ChatGPT

8. Glossari

Active prompting: no existeix una traducció exacta per a aquest terme, però és una metodologia que consisteix a incentivar el pensament crític d'un LLM per millorar la seva capacitat de raonament a través d'interactuar amb ell.

ADT: *Abstract Data Type* en anglès. És la representació d'un model a partir de diversos mètodes que defineixen un comportament concret. Per exemple, el d'una llista enllaçada.

API: *Application Programming Interface* en anglès. Programari que permet establir una comunicació entre dos programes.

AWS: Amazon Web Services. És un servei que ofereix l'empresa Amazon on es poden llogar servidors. Ofereix molta escalabilitat i disposa de productes molt concrets per tasques específiques com administració de bases de dades, aprenentatge computacional, etc.

Bug: error en anglès, especialment relacionat en el món del programari.

Chain-of-thought: cadena de pensament en anglès, és una metodologia que permet millorar la capacitat de raonament d'un LLM a partir de generar passos intermedis que el facin arribar a una resposta correcta.

ChatGPT: intel·ligència artificial preentrenada desenvolupada per l'empresa OpenAI en format de xatbot que utilitza un LLM per donar resposta a les consultes dels usuaris.

Context menu: és una interfície que conté diverses opcions que apareix quan l'usuari interactua amb un programari, normalment amb el botó dret del ratolí. Les opcions més comunes són la de copiar i enganxar.

Flask: és un *framework* que permet la creació d'aplicacions web basades en Python. A més, és eficient a l'hora de crear API, ja que ofereix mecanismes que creen l'entorn necessari sense que el programari s'hagi de preocupar.

Framework: és un entorn de treball que inclou múltiples eines ja preparades amb l'objectiu de minimitzar i generalitzar el treball d'un desenvolupador de codi.

Getter: mètode d'una classe utilitzat per retornar el valor d'un atribut.

GitHub Copilot: intel·ligència artificial preentrenada desenvolupada per les empreses GitHub i OpenAI que permet autocompletar codi font en funció de l'inici d'aquest utilitzant un LLM.

IA: intel·ligència artificial. És la combinació d'algorismes utilitzats d'una determinada manera per oferir capacitats intel·ligents com la tasca de classificar, generar text, etc.

IDE: *Integrated Development Environment*. Programari que ofereix un editor de codi font i facilitats pels programadors de programari. A més, també inclou eines de compilació i de debug.

Java: és un llenguatge de programació orientat a objectes d'alt nivell. Té la peculiaritat de poder-se executar a qualsevol plataforma, és a dir, és multiplataforma. Se sol utilitzar per crear aplicacions empresarials, aplicacions mòbils, dins del desenvolupament web.

Javadoc: és una eina que genera documentació en format HTML de manera automàtica. En ella, s'afegeixen els comentaris del codi i tota la informació referent a la classe: nom, herències, implementacions, mètodes, atributs, etc.

JavaScript: és un llenguatge de programació interpretat d'alt nivell utilitzat principalment dins del desenvolupament web. Tot i això, també s'utilitza en altres camps i també existeixen entorns de treball basats en aquest llenguatge.

Langchain: biblioteca basada en Python utilitzada per a la creació d'una cadena amb el LLM.

Línia d'ordres: és una interfície que permet executar ordres per un usuari sobre un computador o un programa.

LLM: *Large Language Model* en anglès. És un graf entrenat amb múltiples capes que permet generat text a partir de llenguatge natural introduït per un usuari.

Outlier: valor d'un conjunt de dades que difereix considerablement dels altres. Normalment acostumen a representar casos aïllats produïts per errors en la mesura.

Paper: la traducció directa al català és paper. És una publicació acadèmica produïda per una o diverses persones expertes en un camp d'estudi concret on es presenten els resultats i les conclusions d'una recerca.

Plugin: és un component que afegeix una funcionalitat concreta dins d'un programari ja existent.

POST: és un mètode que permet enviar informació quan es duu a terme una consulta. És a dir, permet l'enviament de dades per part d'un usuari o un servidor cap a un altre.

Python: és un llenguatge de programació interpretat d'alt nivell especialment utilitzat en camps com la intel·ligència artificial, tot i que també s'utilitza en altres camps com el desenvolupament web.

Setter: mètode d'una classe utilitzat per assignar un valor a un atribut.

Spam: enviament massiu de missatge no desitjats.

TDD: *Test Driven Development* en anglès. És una metodologia utilitzada en el desenvolupament de programari on, en una primera instància, es desenvolupa una col·lecció de tests i, després, tot el desenvolupament del codi font està guiat per aquests.

Timeout: és l'esdeveniment que ocorre quan un procés no finalitza dins del temps esperat. Està creat amb l'objectiu de no tenir processos que no acaben mai o evitar l'ús de recursos excessius.

UML: *Unified Modeling Language* en anglès. És un llenguatge unificat que persegueix poder representar el disseny d'un sistema.

VPN: *Virtual Private Network* en anglès, és una xarxa virtual privada que ofereix la capacitat d'ocultar l'adreça pública de l'usuari.

9. Bibliografia

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

[2] **A. Radford, K. Narasimhan, T. Salimans i 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 maig del 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 i B. Ge**, *"Summary of ChatGPT/GPT-4 Research and Perspective Towards the Future of Large Language Models"*, 11 de maig del 2023, https://arxiv.org/pdf/2304.01852.pdf (última consulta: 31 de maig del 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 i Y. Zhuangb.** *"HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face"*, 25 de maig del 2023, https://arxiv.org/pdf/2303.17580.pdf (última consulta: 10 de juny del 2023).

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

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

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

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

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

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

[12] **GitHub.** *"Quickstart for GitHub Copilot"*, sense data de publicació, https://docs.github.com/en/copilot/quickstart (última consulta: 31 de maig del 2023).

[13] **OpenAI.** *"Code completion"*, sense data de publicació, https://platform.openai.com/docs/guides/code (última consulta: 31 de maig del 2023).

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

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

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

[17] **Langchain.** *"Introduction"*, sense data de publicació, https://python.langchain.com/docs/get_started/introduction.html (última consulta: 31 de maig del 2023).

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

10. Annexos

11.1. Annex I: Taula de fites inicials

Nom de la fita	Inici	Fi
PAC 0		
Recerca state of art	01/03/23	06/03/23
Redacció proposta	07/03/23	13/03/23
PAC 1		
Creació del diagrama de Gantt	14/03/23	20/03/23
Redacció del pla	21/03/23	28/03/23
PAC 2		
Preparació de l'entorn	29/03/23	29/03/23
Disseny del prototip	30/03/23	01/04/23
Desenvolupament del prototip	03/04/23	05/04/23
Automatització amb *javadoc* de la biblioteca de DED	06/04/23	11/04/23
Automatització amb *javadoc* + tests de la biblioteca de DED	12/04/23	17/04/23
Automatització de creació de tests amb *javadoc* de la biblioteca de DED	18/04/23	22/04/23
Anàlisis i comparació dels primers resultats	24/04/23	29/04/23
Redacció de l'informe	01/05/23	03/05/23
PAC 3		
Implementació de la metodologia *chain-of-thought* al prototip	04/05/23	06/05/23
Implementació de la metodologia *active prompting* al prototip	08/05/23	10/05/23
Anàlisis i comparació de tots els resultats	11/05/23	15/05/23
Integració amb Visual Studio Code	16/05/23	22/05/23
Afegir feedback sobre tests no superats	23/05/23	25/05/23
Redacció de l'informe	26/05/23	29/05/23
PAC 4		
Redacció dels continguts	30/05/23	16/06/23
Revisió final	17/06/23	20/06/23

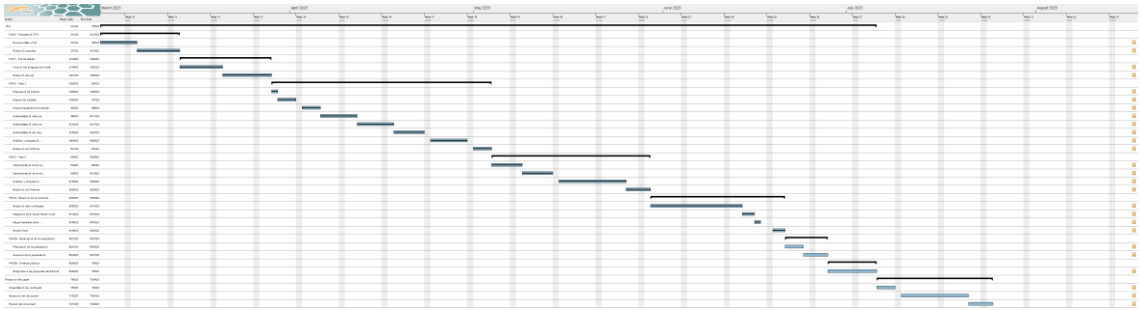
Nom de la fita	Inici	Fi
PAC 5a		
Preparació de la presentació	21/06/23	23/06/23
Gravació de la presentació	24/06/23	27/06/23
PAC 5b		
Respondre a les preguntes del tribunal	28/06/23	05/07/23
Redacció del paper		
Organització de continguts	06/07/23	08/07/23
Redacció del document	10/07/23	20/07/23
Revisió del document	21/07/23	24/07/23

11.2. Annex II: Taula de fites finals

Nom de la fita	Inici	Fi
PAC 0		
Recerca state of art	01/03/23	06/03/23
Redacció proposta	07/03/23	13/03/23
PAC 1		
Creació del diagrama de Gantt	14/03/23	20/03/23
Redacció del pla	21/03/23	28/03/23
PAC 2		
Preparació de l'entorn	29/03/23	29/03/23
Disseny del prototip	30/03/23	01/04/23
Desenvolupament del prototip	03/04/23	05/04/23
Automatització amb *javadoc* de la biblioteca de DED	06/04/23	11/04/23
Automatització amb *javadoc* + tests de la biblioteca de DED	12/04/23	17/04/23
Automatització de creació de tests amb *javadoc* de la biblioteca de DED	18/04/23	22/04/23
Anàlisis i comparació dels primers resultats	24/04/23	29/04/23
Redacció de l'informe	01/05/23	03/05/23
PAC 3		
Implementació d'una cadena per poder enviar exemples de codi	04/05/23	08/05/23
Implementació de la metodologia *active prompting* al prototip	09/05/23	13/05/23
Anàlisis i comparació de tots els resultats	15/05/23	25/05/23
Redacció de l'informe	26/05/23	29/05/23

Nom de la fita	Inici	Fi
PAC 4		
Redacció dels continguts	30/05/23	13/06/23
Integració amb Visual Studio Code	14/06/23	15/06/23
Afegir feedback sobre tests no superats	16/06/23	16/06/23
Revisió final	17/06/23	20/06/23
PAC 5a		
Preparació de la presentació	21/06/23	23/06/23
Gravació de la presentació	24/06/23	27/06/23
PAC 5b		
Respondre a les preguntes del tribunal	28/06/23	05/07/23
Redacció del paper		
Organització de continguts	06/07/23	08/07/23
Redacció del document	10/07/23	20/07/23
Revisió del document	21/07/23	24/07/23

11.3. Annex III. Diagrama de Gantt del treball



Com que la imatge té unes dimensions superiors a l'amplada d'aquest document, s'ha penjat la imatge a Google Drive públicament. La imatge por ser consultada des del següent enllaç: imatge del diagrama de Gantt.

Addicionalment, s'ha penjat la versió web que permet exportar el programa GanttProject en un lloc web personal per poder visualitzar-la en aquest format: diagrama de Gantt en format HTML.

11.4. Annex IV. Repositori del codi font del prototip

El repositori on resideix el codi del prototip s'ha pujat a GitHub de manera pública. S'hi pot accedir mitjançant el següent enllaç: repositori del prototip.

11.5. Annex V. Diagrama UML del prototip

Com que el diagrama UML del prototip té unes dimensions més grans que l'amplada del document, s'ha penjat una captura d'aquest en el següent enllaç: diagrama UML del prototip.

11.6. Annex VI. Repositori de gràfiques

Per consultar totes les gràfiques generades es pot accedir a través del següent enllaç: repositori de gràfiques.

11.7. Annex VII. Repositori de l'extensió de Visual Studio Code

El repositori on resideix el codi de l'extensió per Visual Studio Code s'ha pujat a GitHub de manera pública. S'hi pot accedir mitjançant el següent enllaç: repositori de l'extensió.