UNIVERSIDAD POLITÉCNICA SALESIANA ECUADOR

POSGRADOS

Maestría en
Software
RPC-SO-34-NO.778-2021

Opción de Titulación:
Proyecto de titulación con componentes de investigación aplicada y/o de desarrollo

Tema:
Implementación de un proceso iterativo de entrenamiento para un nuevo modelo en OpenAI que utilice PlantUML, usando lenguaje natural para la generación inteligente de diagramas UML que optimicen los tiempos de diseño en el ciclo de vida de software

Autores:
Ana Belén Arízaga Jaramillo
Wilson Andrés Arízaga Jaramillo

Director:
Christian Mauricio Merchán Millán

Cuenca – Ecuador
2025

Autores:

Ana Belén Arízaga Jaramillo
Ingeniero de Sistemas.
Candidato a Magíster en Software por la Universidad Politécnica Salesiana – Sede Cuenca.
aarizagaj@est.ups.edu.ec

Wilson Andrés Arízaga Jaramillo
Ingeniero de Sistemas.
Candidato a Magíster en Software por la Universidad Politécnica Salesiana – Sede Cuenca.
warizagaj@est.ups.edu.ec

Dirigido por:

Christian Mauricio Merchán Millán
Ingeniero en Computación.
Magíster en Sistemas de Información.
cmerchan@ups.edu.ec

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra para fines comerciales, sin contar con autorización de los titulares de propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual. Se permite la libre difusión de este texto con fines académicos investigativos por cualquier medio, con la debida notificación a los autores.

DERECHOS RESERVADOS
2025© Universidad Politécnica Salesiana.
CUENCA – ECUADOR – SUDAMÉRICA
ANA BELÉN ARÍZAGA JARAMILLO
WILSON ANDRÉS ARÍZAGA JARAMILLO
Implementación de un proceso iterativo de entrenamiento para un nuevo modelo en Openai que utilice Plantuml, usando lenguaje natural para la generación inteligente de diagramas UML que optimicen los tiempos de diseño en el ciclo de vida de software

DEDICATORIA

Como seres humanos, en algún punto nos damos cuenta qué el camino de la vida es un viaje único el cual lo recorremos con personas pasajeras, otras eternas, y las más importantes que son la familia, por lo que siempre van a ser nuestro pilar para seguir superándonos cada día. Esta etapa se la dedico a mi esposa, que siempre está ahí para mí y ha sido mi compañera de este viaje con su amor incondicional, mis hijos que me llenan el corazón cuando vienen a mí con sus preguntas, mis padres que me han criado como una buena persona con su ejemplo diario.

Wilson Andrés Arízaga Jaramillo

AGRADECIMIENTO

Primero, el agradecimiento a Dios por estar siempre a mi lado en todo momento y darme la motivación para seguir adelante; a mi esposa, mis hijos y mi familia que siempre me han apoyado en el proceso de este nuevo reto, que con su sonrisa y cariño han estado siempre ahí. A mi hermana Belén por haber sido mi equipo y apoyo durante toda esta nueva aventura, que nos ha llevado juntos a conseguir un nuevo logro en nuestras vidas.

A mis compañeros de trabajo en la Cooperativa Jardín Azuayo, por su compañerismo y disposición para ayudar en todo momento.

A mi tutor de tesis, Msc. Christian Merchán, por su invaluable guía y apoyo durante el desarrollo de este proyecto. Su dedicación, conocimiento y constante disposición para orientarme en cada etapa de la investigación fueron esenciales para el logro de mis objetivos.

Gracias por su compromiso con la excelencia académica y por crear un entorno propicio para el crecimiento y la formación.

Wilson Andrés Arízaga Jaramillo

DEDICATORIA

Dedico este trabajo a Dios, cuya presencia y bendición han sido mi fortaleza y guía en cada paso de esta etapa. Agradezco la fe que me has dado, y por los cambios reales y significativos que has obrado en mi vida. Dedico también este esfuerzo a la familia que me has regalado, quienes siempre han sido comprensivos y alentadores.

A mi esposo Iván y a nuestros hijos Rafa y Emi, quienes han sido mi motivación constante y el motor de este logro. A mis padres, Wilson y Ana, a mis hermanos, Andrés y Johanna, por su apoyo y cariño incondicional. A mi cuñada Alex quien siempre me ha demostrado el poder de las mujeres, gracias por creer en mí y empoderarme.

Ana Belén Arízaga Jaramillo

AGRADECIMIENTO

Quiero comenzar expresando mi profundo agradecimiento a todas las personas que han sido fundamentales en el logro de este objetivo. En primer lugar, a mi esposo Iván, quien ha sido mi constante apoyo y motivación, impulsándome siempre a superarme y crecer en cada paso de este camino. A mis hijos Rafa y Emi, cuyo cariño y comprensión han sido mi mayor aliento. A pesar de su corta edad, han mostrado un espíritu generoso y dispuesto, brindándome su apoyo y entendimiento en esta etapa de crecimiento personal.

A mis padres, quienes han estado a mi lado desde mis primeros pasos, alentándome con su amor, apoyo y sabias palabras. Gracias por ser mis guías y por siempre motivarme a avanzar y alcanzar nuevas metas.

A mi hermano Andrés, quien no solo me animó a embarcarme en esta maestría, sino que también ha estado presente en cada momento importante. Ha sido un honor haber realizado este trabajo de titulación juntos, compartiendo el esfuerzo y la satisfacción de este logro.

Agradezco también a todos los profesionales que conocí durante la maestría: docentes, funcionarios y compañeros, por su ayuda y apoyo constante. Especialmente, quiero expresar mi sincero agradecimiento al Msc. Christian Merchán, por su tiempo, dedicación y por compartir generosamente su conocimiento y experiencia guiando este trabajo de titulación. Gracias a su orientación y disposición, logramos construir juntos un trabajo que refleja todo lo aprendido y alcanzado durante esta etapa.

Este logro no hubiera sido posible sin la bendición de Dios, quien me ha acompañado y guiado en cada paso, haciéndome sentir Su presencia en momentos clave de mi vida.

Ana Belén Arízaga Jaramillo

Tabla de Contenido

Resumen 12
Abstract 13
1. Introducción 14
1.1. Determinación del Problema 15
1.2. Alcance 16
1.3. Justificación 17
1.4. Objetivos 18
1.4.1. Objetivo General 18
1.4.2. Objetivos Específicos 18
2. Marco teórico referencial 21
2.1. Fundamentos de la Inteligencia Artificial y Aprendizaje Automático 22
2.2. Inteligencia Artificial (IA) 23
2.2.1. Aprendizaje Automático (AA) 23
2.2.2. Aprendizaje Profundo (Deep Learning) 24
2.2.3. Historia y evolución 25
2.3. Procesamiento del Lenguaje Natural (PLN) 26
2.3.1. Técnicas y modelos 27
2.3.2. Aplicaciones de PLN en la generación de código y documentación 28
2.4. OpenAI y sus Modelos Generativos 29
2.4.1. Metodologías de entrenamiento 30
2.4.2. Fine-Tuning (Ajuste Fino) en Modelos de Lenguaje Natural 31
2.5. Lenguaje de Modelado Unificado (UML) 33
2.5.1. PlantUML 34
2.5.2. Generación de Diagramas UML a través de PlantUML 35
2.6. APIs y Microservicios 36
2.6.1. Diseño e Implementación de APIs 37
2.6.2. Integración de Microservicios 38
2.7. Integración de Chatbots en Entornos de Desarrollo 39
2.7.1. Funcionalidad y Usabilidad del Chatbot de Telegram 40
2.7.2. Interacción en Tiempo Real para la Generación de Diagramas UML 41
2.8. Ciclo de Vida del Desarrollo de Software 42

2.8.1. Impacto de la Automatización en el Ciclo de Vida 43
2.8.2. Optimización de Tiempos de Diseño 44
2.9. Estudios de Caso y Aplicaciones Relevantes 46
2.10. Retos y consideraciones 48
2.11. Relación entre el marco teórico y la metodología 50
3. Desarrollo de La Solución 52
3.1. Metodología 52
3.2. Capacidades ofrecidas por OpenAI 55
3.3. Funcionamiento de PlantUML 64
3.4. Entrenamiento y evaluación del Modelo 69
3.5. Arquitectura de la solución 80
3.6. Desarrollo de una api integradora 87
3.7. Evaluación del Comportamiento del Modelo a Través de la Interfaz 96
4. Resultados y discusión 110
4.1 Evaluación y mejora a través de las versiones de modelos fine - tuned 110
5. Conclusiones 112
Referencias 114

Tabla de Ilustraciones

FIGURA 1. DIAGRAMA DE INTELIGENCIA ARTIFICIAL (BENAVIDES ET AL., 2023) 23
FIGURA 2. EJEMPLO DE FLUJO DE ENTRENAMIENTO DE UN SISTEMA DE PROCESAMIENTO DE TEXTO BASADO EN APRENDIZAJE AUTOMÁTICO (PRIORE ET AL., 2001) 24
FIGURA 3. ESQUEMA DE RECONOCIMIENTO DE LA IMAGEN DE UN PERRO (WOLCHOVER, 2020) 25
FIGURA 4. HISTORIA CONTEMPORÁNEA DEL PLN (DIAZ & DIAZ, 2023) 26
FIGURA 5. PROCESO DE PREPARACIÓN DEL TEXTO RECIBE EL NOMBRE DE CONSTRUCCIÓN DEL CORPUS (DIAZ & DIAZ, 2023) 27
FIGURA 6. MODELOS DE PROCESAMIENTO DE LENGUAJE NATURAL (ELABORACIÓN PROPIA) 28
FIGURA 7. COPILOTO DE GITHUB QUE UTILIZA OPENAI CODEX PARA CODIFICACIÓN (GITHUB) 29
FIGURA 8. EVOLUCIÓN DE LA SERIE CHAT GPT (HUSSAIN, 2023) 30
FIGURA 9. MODALIDAD DE ENTRENAMIENTO DE CHATGPT DE OPENAI (OPENAI, N.D.) 31
FIGURA 10. DIAGRAMA DE FUNCIONAMIENTO DEL MODELO AFINADO (FINED-TUNED MODEL) (OPENAI, N.D.) 32
FIGURA 11. MODELO UML (ELABORACIÓN PROPIA) 33
FIGURA 12. CÓDIGO Y DIAGRAMA PLANTUML (PLANTUML WEBSITE, N.D.) 35
FIGURA 13. DIAGRAMAS UML (ELABORACIÓN PROPIA) 36
FIGURA 14. FLUJO DE PETICIÓN DE INFORMACIÓN A TRAVÉS DE UNA API (AGUILERA, 2024) 38
FIGURA 15. APLICACIÓN WEB DE UNA TIENDA CON MICROSERVICIOS (ATLASSIAN, N.D.) 39
FIGURA 16. ARQUITECTURA DE UN CHATBOT (COMPRENDIENDO LA ARQUITECTURA DE LOS CHATBOTS, N.D.) 40
FIGURA 17. FLUJO DE INFORMACIÓN DE CHATBOT CON TELEGRAM (ÁLVARO, 2017) 41
FIGURA 18. INTEGRACIÓN APIS CON TELEGRAM (AVUTHU, 2023) 42
FIGURA 19. CICLO DE VIDA DE DESARROLLO DE APLICACIONES (TRPLANE, 2021) 43
FIGURA 20. AUTOMATIZACIÓN DEL CICLO DE VIDA SOFTWARE (ADMIN, 2023) 44
FIGURA 21. MÉTRICA DE RENDIMIENTO DE ENTREGA DE SOFTWARE (ACCELERATE STATE OF DEVOPS 2021 | GOOGLE CLOUD, N.D.) 45
FIGURA 22. CICLO DE LA AUTOMATIZACIÓN DE PROCESOS (BELANDRIA, 2023) 46
FIGURA 23. INTEGRACIÓN DE UN CHATBOT CON EL ASISTENTE IBM WATSON (IBM WATSON ASSISTANT, TELEGRAM AND PYTHON, N.D.) 48
FIGURA 24. DIAGRAMA DE FLUJO DE OPENAI Y EL PROCESAMIENTO DEL LENGUAJE NATURAL (NLP) 57
FIGURA 25. DIAGRAMA DE FLUJO DE PLANTUML Y SU API (ELABORACIÓN PROPIA) 66
FIGURA 26. DIAGRAMA DE CLASES EN PLANTUML (ELABORACIÓN PROPIA) 69
FIGURA 27. DIAGRAMA DE FINE TUNING Y ENTRENAMIENTO DEL MODELO (ELABORACIÓN PROPIA) 70
FIGURA 28. CONFIGURACIÓN DE API KEYS EN OPENAI (OPENAI, N.D.) 71
FIGURA 29. CARGA DE ARCHIVOS (OPENAI, N.D.) 73
FIGURA 30. SELECCIONADOR DE ARCHIVOS DE ENTRENAMIENTO (OPENAI, N.D.) 73
FIGURA 31. ARCHIVOS DE ENTRENAMIENTO CARGADOS (OPENAI, N.D.) 74
FIGURA 32. PANTALLA DE FINE TUNING (OPENAI, N.D.) 74
FIGURA 33. CONFIGURACIÓN DE FINE TUNING SELECCIONANDO UN MODELO 75
FIGURA 34. PANTALLA DETALLE DE CREACIÓN DE FINE TUNING (OPENAI, N.D.) 75
FIGURA 35. MODELOS FINE-TUNING CREADOS EXITOSAMENTE (OPENAI, N.D.) 76
FIGURA 36. TABLA COMPARATIVA DE MODELOS GENERADOS CON FINE-TUNING (ELABORACIÓN PROPIA) 76
FIGURA 37. ANÁLISIS MODELOS FINE TUNING (ELABORACIÓN PROPIA) 79
FIGURA 38. ESCENARIO DEL SISTEMA (ELABORACIÓN PROPIA) 81
FIGURA 39. CONTEXTO DEL SISTEMA CHATBOT DE TELEGRAM (ELABORACIÓN PROPIA) 82
FIGURA 40. DIAGRAMA DE CONTENEDORES Y RELACIONES DE LA SOLUCIÓN (ELABORACIÓN PROPIA) 83
FIGURA 41. DIAGRAMA DE COMPONENTES DEL BACKEND (ELABORACIÓN PROPIA) 84
FIGURA 42. DIAGRAMA DE COMPONENTES DEL API DE INTEGRACIÓN (ELABORACIÓN PROPIA) 84
FIGURA 43. DIAGRAMA DE DESPLIEGUE EN AMBIENTE DE DESARROLLO (ELABORACIÓN PROPIA) 86
FIGURA 44. DIAGRAMA DE DESPLIEGUE EN AMBIENTE DE PRODUCCIÓN (ELABORACIÓN PROPIA) 87
FIGURA 45. APLICACIÓN BOTFATHER EN TELEGRAM (ELABORACIÓN PROPIA) 88
FIGURA 46. CONFIGURACIÓN PARA CREAR EL NUEVO CHATBOT (ELABORACIÓN PROPIA) 88

FIGURA 47. TOKEN DEVUELTO PARA UTILIZAR POR UNA API Y LA URL DEL CHATBOT CREADO (ELABORACIÓN PROPIA) 89
FIGURA 48. PLANTUML SERVER REPO PARA GENERAR DIAGRAMAS UML (PLANTUML WEBSITE, N.D.) ..89
FIGURA 49. SERVIDOR PLANTUML LEVANTADO EN DOCKER (ELABORACIÓN PROPIA) 90
FIGURA 50. ESTRUCTURA DEL PROYECTO DE INTEGRACIÓN (ELABORACIÓN PROPIA) 90
FIGURA 51. API KEYS DE OPENAI, TELEGRAM Y MODELO ENTRENADO (ELABORACIÓN PROPIA) 90
FIGURA 52. EJEMPLO DE FUNCIONAMIENTO DE LA SOLUCIÓN (ELABORACIÓN PROPIA) 95
FIGURA 53. RESPUESTA DE SOLICITUD PARA DIAGRAMA UML1 98
FIGURA 54. RESPUESTA DE SOLICITUD MANUAL PARA DIAGRAMA UML1 98
FIGURA 55. RESPUESTA DE SOLICITUD PARA DIAGRAMA UML2 98
FIGURA 56. RESPUESTA DE SOLICITUD MANUAL PARA DIAGRAMA UML2 98
FIGURA 57. RESPUESTA DE SOLICITUD PARA DIAGRAMA UML3 99
FIGURA 58. RESPUESTA DE SOLICITUD MANUAL PARA DIAGRAMA UML3 99
FIGURA 59. RESPUESTA DE SOLICITUD PARA DIAGRAMA UML4 100
FIGURA 60. RESPUESTA DE SOLICITUD MANUAL PARA DIAGRAMA UML4 100
FIGURA 61. RESPUESTA DE SOLICITUD PARA DIAGRAMA UML5 100
FIGURA 62. RESPUESTA DE SOLICITUD MANUAL PARA DIAGRAMA UML5 100
FIGURA 63. RESPUESTA DE SOLICITUD PARA DIAGRAMA UML6 101
FIGURA 64. RESPUESTA DE SOLICITUD MANUAL PARA DIAGRAMA UML6 101
FIGURA 65. RESPUESTA DE SOLICITUD PARA DIAGRAMA UML7 101
FIGURA 66. RESPUESTA DE SOLICITUD MANUAL PARA DIAGRAMA UML7 101
FIGURA 67. RESPUESTA DE SOLICITUD PARA DIAGRAMA UML8 102
FIGURA 68. RESPUESTA DE SOLICITUD MANUAL PARA DIAGRAMA UML8 102
FIGURA 69. MENSAJE DE ERROR DE LA SOLUCIÓN CUANDO SUPERÓ EL NÚMERO DE TOKENS (ELABORACIÓN PROPIA) 103
FIGURA 70. EJEMPLO DE SOLICITUD PARA DIAGRAMA UML CON LA VERSIÓN DEL MODELO 1 (ELABORACIÓN PROPIA) 104
FIGURA 71. EJEMPLO DE SOLICITUD PARA DIAGRAMA UML CON LA VERSIÓN DEL MODELO 2 (ELABORACIÓN PROPIA) 105
FIGURA 72. EJEMPLO DE SOLICITUD PARA DIAGRAMA UML CON LA VERSIÓN DEL MODELO 3 (ELABORACIÓN PROPIA) 106
FIGURA 73. EJEMPLO DE SOLICITUD PARA DIAGRAMA UML CON LA VERSIÓN DEL MODELO 4 (ELABORACIÓN PROPIA) 107
FIGURA 74. EJEMPLO DE SOLICITUD PARA DIAGRAMA UML CON LA VERSIÓN DEL MODELO 5 (ELABORACIÓN PROPIA) 108

IMPLEMENTACIÓN DE UN PROCESO ITERATIVO DE ENTRENAMIENTO PARA UN NUEVO MODELO EN OPENAI QUE UTILICE PLANTUML, USANDO LENGUAJE NATURAL PARA LA GENERACIÓN INTELIGENTE DE DIAGRAMAS UML QUE OPTIMICEN LOS TIEMPOS DE DISEÑO EN EL CICLO DE VIDA DE SOFTWARE.

AUTOR(ES):

ANA BELÉN ARÍZAGA JARAMILLO
WILSON ANDRÉS ARÍZAGA JARAMILLO

Resumen

Este trabajo explora la creación de una solución integrada que automatiza la generación de diagramas de tipo UML a partir de descripciones en lenguaje natural, utilizando un modelo de inteligencia artificial basado en OpenAI y la herramienta de diagramación PlantUML. A través de un proceso iterativo de entrenamiento, el modelo se adapta para interpretar descripciones textuales específicas del dominio de desarrollo de software, generando diagramas UML precisos y en conformidad con las especificaciones. La integración de un chatbot de Telegram permite a los desarrolladores interactuar con el sistema desde dispositivos móviles o estaciones de trabajo, mejorando la accesibilidad y la eficiencia en el proceso de diseño. La metodología incluye un análisis detallado del rendimiento del modelo en procesamiento de lenguaje natural (NLP), optimización de su precisión para la generación de diversos tipos de diagramas UML, y el desarrollo de una API de microservicios que facilita la comunicación entre el modelo de IA, PlantUML y Telegram. Los resultados de las pruebas demuestran que el sistema cumple con el objetivo de reducir tiempos y errores en el diseño, proporcionando una herramienta adaptable y eficiente para entornos de desarrollo ágiles.

Palabras clave:

OpenAI, PlantUML, lenguaje natural, entrenamiento iterativo, diagramas UML, API, chatbot, Telegram, microservicios.

Abstract

This study explores the development of an integrated solution that automates the generation of UML diagrams from natural language descriptions using an OpenAI-based artificial intelligence model and the PlantUML diagramming tool. Through an iterative training process, the model is tailored to interpret textual descriptions relevant to software development, generating accurate and specification-compliant UML diagrams. Telegram chatbot integration allows developers to interact with the system from mobile devices or workstations, enhancing accessibility and efficiency in the design process. The methodology includes a detailed analysis of the model's natural language processing (NLP) performance, optimizing its accuracy for generating various UML diagram types, and developing a microservices-based API to facilitate communication between the AI model, PlantUML, and Telegram. Test results demonstrate that the system effectively reduces design time and errors, providing an adaptable, efficient tool suited for agile development environments.

Key words:
OpenAI, PlantUML, natural language, iterative training, UML diagrams, API, chatbot, Telegram, microservices, agile development.

1. Introducción

La evolución tecnológica ha impactado de manera significativa en el desarrollo de software, promoviendo la implementación de herramientas que mejoran los procesos de diseño y gestión de sistemas. La automatización juega un rol esencial en optimizar la eficiencia, sobre todo en la creación de diagramas UML, los cuales son cruciales para la representación visual de sistemas dentro del ciclo de vida del software. En este contexto, los avances en inteligencia artificial, como los modelos generativos de OpenAI, ofrecen una alternativa innovadora para automatizar la creación de diagramas a partir de descripciones en lenguaje natural.

Modelos como GPT de OpenAI, entrenados con grandes volúmenes de datos, han demostrado una capacidad destacada para convertir lenguaje natural en representaciones visuales, lo que acelera notablemente los procesos de diseño. Estas funcionalidades introducen una nueva etapa de automatización en el desarrollo de software. Por su parte, PlantUML, una herramienta de software libre destinada a la creación de diagramas UML desde texto, ha obtenido reconocimiento por su facilidad de uso y eficacia. Utilizada en numerosos proyectos, esta herramienta mejora la comunicación técnica en equipos de desarrollo al facilitar la creación de diagramas.

Este proyecto propone una solución que no solo acelera los tiempos de diseño, sino que también mejora la precisión y reduce errores comunes en la creación de diagramas UML. Desde una perspectiva industrial, la capacidad de generar diagramas automáticamente a partir de descripciones en lenguaje natural permite a las empresas reducir costos operativos y optimizar recursos al eliminar tareas repetitivas y propensas a errores. En el ámbito académico, esta investigación contribuye significativamente al estudio del procesamiento de lenguaje natural (PLN) aplicado a la ingeniería de software, proporcionando un caso de uso práctico que puede ser replicado o mejorado en futuras investigaciones. Además, la integración de un chatbot como interfaz amigable democratiza el acceso a

herramientas avanzadas, permitiendo la participación de stakeholders con distintos niveles de experiencia técnica. Esta combinación de accesibilidad y automatización representa un avance relevante tanto para la academia como para la industria.

Al final de esta sección, es importante resaltar la necesidad de un marco teórico sólido que sirva de base para las decisiones metodológicas adoptadas. El siguiente capítulo abordará los fundamentos teóricos que sustentan este trabajo y su relevancia en el contexto del desarrollo de software automatizado.

1.1. Determinación del Problema

El núcleo del problema se centra en la necesidad de acelerar la etapa de diseño en la evolución de desarrollo de software mediante la generación automática de diagramas UML, un recurso visual crucial en la representación de la arquitectura de un sistema. Los diagramas UML facilitan la comprensión y la colaboración en equipos de desarrollo (Booch, Rumbaugh, & Jacobson, 2005). Sin embargo, la generación y mantenimiento de estos diagramas de forma manual es un proceso que consume tiempo y no se adapta bien a metodologías ágiles, en las que los requisitos cambian frecuentemente, lo cual puede ocasionar retrasos y aumentar la probabilidad de errores.

Aunque algunos autores han destacado la relevancia de los diagramas UML en las etapas del software, la generación manual de estos resulta ineficaz en contextos que demandan agilidad (Jacobson, 2004; Schwaber & Sutherland, 2020). Existen herramientas de automatización, pero muchas de ellas enfrentan dificultades de integración en entornos de desarrollo y no aprovechan la capacidad del procesamiento de lenguaje natural (NLP), restringiendo de este modo su potencial de optimizar el diseño y reducir errores (OMG, 2017).

La inteligencia artificial ha mostrado avances significativos, especialmente con modelos como GPT-3 y GPT-4 de OpenAI, que brindan funcionalidades sobresalientes para automatizar tareas mediante lenguaje cotidiano (Brown et al., 2020). No obstante, su aplicación en la elaboración de diagramas UML

automatizados aún no ha sido ampliamente investigada, lo cual deja un espacio en la industria donde la automatización podría mejorar la precisión y disminuir los plazos en el desarrollo de software.

La propuesta de combinar un modelo de OpenAI con herramientas como PlantUML ofrece una solución para este problema. Esta combinación podría permitir que los desarrolladores describan sistemas en lenguaje natural y obtengan diagramas UML generados automáticamente, simplificando el proceso. Asimismo, el uso de un chatbot de Telegram como interfaz facilita la accesibilidad, permitiendo interacciones rápidas y resultados inmediatos sin conocimientos técnicos avanzados en modelado gráfico.

El problema principal radica en la necesidad de agilizar el diseño de diagramas UML en metodologías ágiles, minimizando los tiempos y errores asociados. En este contexto, identificar herramientas que aprovechen la inteligencia artificial para automatizar procesos se vuelve crucial. La siguiente sección detallará el alcance del estudio, estableciendo los límites y objetivos concretos que abordan esta problemática.

1.2. Alcance

El alcance de este estudio se enfoca en la ejecución de un proceso iterativo de entrenamiento para un nuevo modelo en OpenAI, el cual utilizará PlantUML con capacidad de interpretar descripciones de lenguaje natural para construir diagramas UML. Este proceso buscará desarrollar y afinar un modelo que no solo entienda el texto descriptivo de los requisitos del software, sino que también lo traduzca eficientemente en diagramas UML precisos y técnicamente viables.

Además, el estudio incluirá la integración de un chatbot de Telegram que permitirá a los usuarios solicitar y recibir diagramas UML de manera inmediata mediante interacciones de comunicación escrita. Este chatbot servirá como una interfaz accesible y fácil de usar para la generación de diagramas, facilitando la participación

de stakeholders no especializados y mejorando la accesibilidad del proceso de diseño.

El modelo desarrollado deberá adaptarse a cambios y mejoras continuas en los requisitos del sistema, asegurando su aplicabilidad en entornos de desarrollo dinámico. La incorporación del chatbot de Telegram tiene como objetivo reducir significativamente los tiempos de diseño en el proceso de desarrollo del software al proporcionar una solución interactiva y eficiente para la generación de diagramas UML, adaptada a las necesidades y cambios constantes del proceso de desarrollo (Booch et al., 2005; Russell & Norvig, 2021).

Este enfoque busca responder a la problemática identificada, garantizando aplicabilidad en entornos ágiles. La próxima sección justificará la relevancia y los beneficios esperados de esta investigación para la industria del software y la academia.

1.3. Justificación

Este proyecto de investigación se justifica en la posibilidad de una transformación significativa que la implementación de este modelo alcanzaría a traer a la industria del software. Al integrar avanzados métodos de procesamiento del lenguaje natural (NLP) y aprendizaje automático para la creación automatizada de diagramas UML, se anticipa una disminución considerable en el tiempo requerido para la fase de diseño. Este es un aspecto crítico para mantener la competitividad en mercados que exigen rapidez y eficiencia. La incorporación de un chatbot de Telegram, que permitirá a los usuarios solicitar y recibir diagramas UML mediante interacciones en lenguaje natural, representa un avance significativo en la accesibilidad y usabilidad del proceso de diseño.

La capacidad del chatbot para facilitar la comunicación directa y eficiente con el sistema de generación de diagramas UML ayudará a superar las barreras del lenguaje técnico, permitiendo una mayor participación de stakeholders no especializados. Esto no solo mejorará la precisión en la fase de diseño, sino que

también reducirá los costos de desarrollo posteriores al minimizar los errores derivados de malentendidos en los requisitos o cambios no documentados adecuadamente.

La investigación proporcionará perspectivas valiosas sobre la incorporación de tecnologías emergentes, como la inteligencia artificial y los chatbots, en las prácticas estándares de desarrollo de software. Al demostrar cómo estas tecnologías logran automatizar y optimizar el proceso de diseño de manera efectiva, este proyecto marcará un paso importante hacia la automatización inteligente en la ingeniería de sistemas y software, estableciendo nuevas oportunidades con el objetivo de aumentar la eficiencia y la calidad en el desarrollo de software (Beck et al., 2001; Russell & Norvig, 2021).

Este proyecto es relevante porque aborda una necesidad clave en el desarrollo ágil de software: la reducción de tiempos y errores en el diseño de diagramas UML. Al incorporar herramientas avanzadas de IA y accesibilidad, se espera generar un impacto significativo tanto en el ámbito académico como en el industrial. La siguiente sección introducirá los objetivos específicos que guían la investigación y definen sus metas claras.

1.4. Objetivos

1.4.1. Objetivo General

Implementar un proceso iterativo de entrenamiento para un nuevo modelo en OpenAI que utilice PlantUML, integrando un chatbot de Telegram para permitir la solicitud y recepción de diagramas UML mediante descripciones en lenguaje natural, con el objetivo de optimizar los tiempos de diseño y mejorar la accesibilidad en el ciclo de vida del desarrollo de software.

1.4.2. Objetivos Específicos

- *Investigar y analizar las capacidades ofrecidas por OpenAI:* Realizar un análisis exhaustivo de las funcionalidades que ofrece OpenAI, evaluando las características y capacidades de sus modelos en el contexto de generación

de diagramas UML. Esto incluye la integración con sistemas de procesamiento de lenguaje natural (NLP) y la preparación para utilizar un chatbot de Telegram en la interacción con los usuarios.

- *Analizar y entender el funcionamiento de PlantUML*: Diseñar y determinar detalladamente los requisitos iniciales y las secuencias de código utilizadas por PlantUML para generar diagramas UML mediante su API. Evaluar cómo esta herramienta puede ser utilizada en combinación con el modelo de OpenAI y el chatbot de Telegram para mejorar la generación y accesibilidad de los diagramas UML.
- *Entrenamiento y evaluación del modelo de OpenAI:* Con las secuencias y datos relevantes identificados, entrenar un modelo optimizado para la generación de diagramas UML, que sea capaz de interpretar descripciones en lenguaje natural proporcionadas por el chatbot de Telegram. Implementar un proceso de evaluación iterativa para afinar el modelo y determinar la calidad y precisión de los diagramas UML generados.
- *Desarrollar una API de integración que utilice las APIs seleccionadas:* Implementar los microservicios necesarios para la integración del modelo de OpenAI y la API de PlantUML, así como la incorporación del chatbot de Telegram. Realizar pruebas exhaustivas para verificar la correcta incorporación de las APIs y asegurar el funcionamiento adecuado del sistema de generación de diagramas UML.
- *Evaluar el comportamiento del nuevo modelo de IA a través de la interfaz de integración:* Realizar pruebas para medir la eficacia del modelo generado en la generación de diagramas UML mediante descripciones en lenguaje natural proporcionadas a través del chatbot de Telegram. Evaluar su nivel de eficiencia y precisión en diferentes casos de uso, y ajustar el sistema según los resultados obtenidos para optimizar la experiencia del usuario y la calidad de los diagramas UML.

2. Marco teórico referencial

La implementación de un proceso iterativo de entrenamiento para un nuevo modelo en OpenAI, destinado a la creación inteligente de diagramas UML a través de PlantUML y lenguaje natural, constituye un progreso significativo en la optimización de tiempos de diseño a lo largo de las etapas del desarrollo de software. Este trabajo se ancla en los fundamentos de la inteligencia artificial (IA) y el aprendizaje automático (AA), explorando específicamente el procesamiento del lenguaje natural (PLN) para traducir descripciones de alto nivel a especificaciones técnicas detalladas.

La integración de un chatbot de Telegram en este proceso añade una capa adicional de accesibilidad y usabilidad, permitiendo a los usuarios una interacción con el sistema de generación de diagramas UML mediante descripciones textuales, y recibir respuestas inmediatas en forma de diagramas visuales. Esta característica es particularmente relevante para facilitar la participación de stakeholders no especializados y para agilizar la comunicación y el diseño en tiempo real.

La relevancia de OpenAI en el ambito del PLN, junto con la capacidad de PlantUML para facilitar la elaboración automatizada de diagramas UML, subraya la capacidad de este trabajo para mejorar significativamente la eficiencia en la ingeniería de software. Al investigar las metodologías de desarrollo de software, la importancia crítica de los diagramas UML en la planificación y documentación de propuestas, y la implementación de interfaces accesibles como el chatbot de Telegram, esta tesis pretende acortar la distancia entre la conceptualización rápida y la implementación efectiva.

A través de un enfoque metodológico que combina teoría y práctica, se busca no solo avanzar en la comprensión teórica de estas tecnologías, sino también demostrar su aplicabilidad y beneficios en escenarios reales de desarrollo de software. La integración del chatbot de Telegram, en particular, es un componente

clave que promete mejorar la interacción usuario-sistema, optimizando el proceso de diseño y adaptándolo a las necesidades dinámicas del desarrollo de software.

2.1. Fundamentos de la Inteligencia Artificial y Aprendizaje Automático

Los conceptos de inteligencia artificial (IA) y aprendizaje automático (AA) constituyen la base de esta investigación, proporcionando el marco necesario para comprender cómo se desarrollan e implementan modelos avanzados en tecnología de software. La IA se centra en imitar capacidades cognitivas humanas a través de sistemas computacionales, mientras que el AA, un campo especializado dentro de la IA, facilita que las máquinas optimicen su desempeño al aprender de los datos, sin depender de una programación explícita. En este estudio, se explora cómo estas disciplinas facilitan la creación de sistemas que interpretan lenguaje natural y generan diagramas UML, enfatizando el impacto de modelos generativos, como los desarrollados por OpenAI, para automatizar y optimizar procesos en la ingeniería de software.

2.2. Inteligencia Artificial (IA)

La inteligencia artificial (IA) se define como el campo que explora el diseño y construcción de sistemas computacionales con la capacidad de ejecutar tareas asociadas típicamente con la capacidad cognitiva humana, como el procesamiento visual, el reconocimiento de voz, la toma de decisiones y la traducción de lenguajes (Russell & Norvig, 2021). La IA se clasifica en dos grandes tipos: la IA débil, que se enfoca en resolver problemas específicos, y la IA fuerte, orientada a replicar la cognición humana de forma general. A lo largo de los años, los avances en IA han permitido crear desde sistemas capaces de superar a los humanos en juegos complejos hasta algoritmos que diagnostican enfermedades con un nivel de precisión similar o mayor a la de expertos. (Kaplan & Haenlein, 2019).

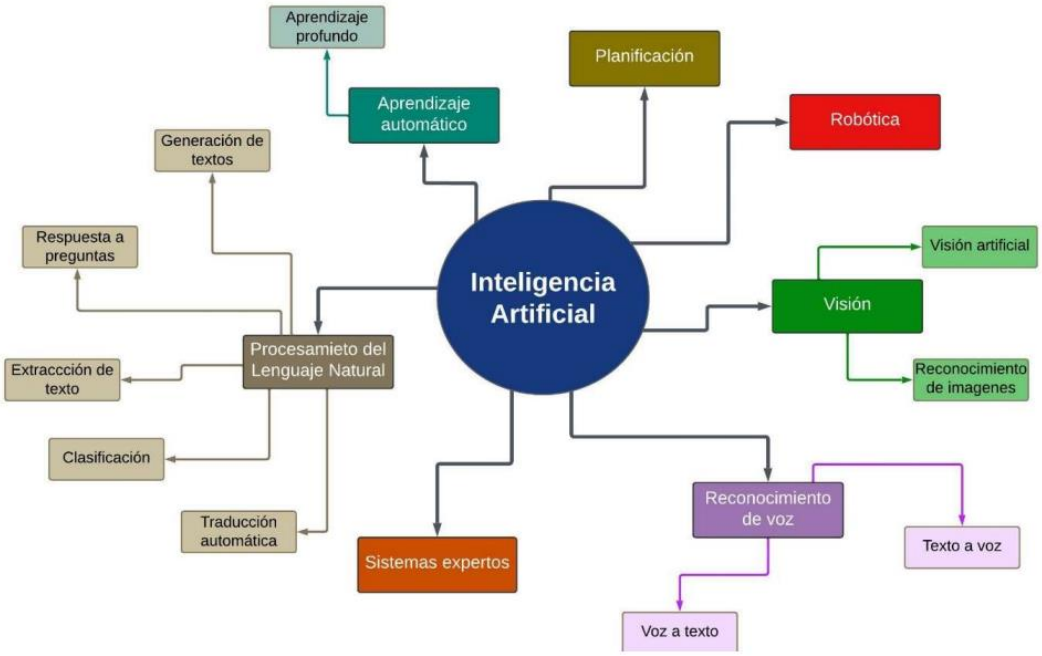


Figura 1. Diagrama de Inteligencia Artificial **(Benavides et al., 2023)**

2.2.1. Aprendizaje Automático (AA)

El aprendizaje automático (AA) es una especialidad dentro del campo de la inteligencia artificial que permite a los sistemas incrementar su eficacia en diversas tareas a través de la experiencia adquirida mediante datos. Esta metodología se fundamenta en la idea de que las máquinas pueden aprender patrones y tomar

decisiones sin requerir programación explícita en cada paso, lo que abre un amplio abanico de aplicaciones en campos que van desde la predicción en finanzas hasta la recomendación personalizada en plataformas de streaming (Murphy, 2012). La importancia del AA radica en su capacidad de adaptarse y optimizar continuamente sus procesos, convirtiéndolo en un elemento esencial en el avance de tecnologías inteligentes. (Jordan & Mitchell, 2015).

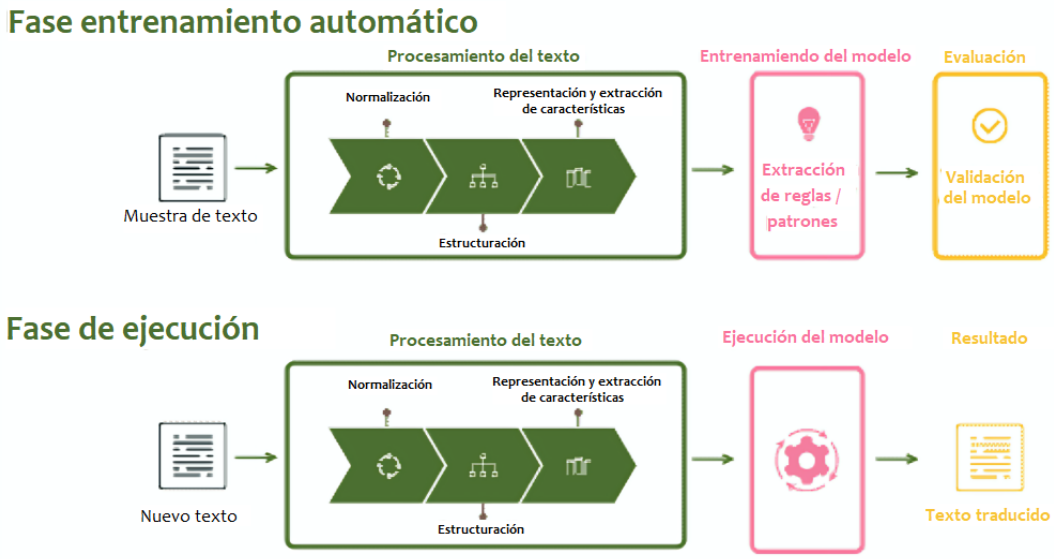


Figura 2. Ejemplo de flujo de entrenamiento de un sistema de procesamiento de texto basado en aprendizaje automático (Priore et al., 2001)

2.2.2. Aprendizaje Profundo (Deep Learning)

El aprendizaje profundo es un método sofisticado dentro del aprendizaje automático que utiliza redes neuronales artificiales dispuestas en varias capas para examinar y analizar patrones complejos en amplios conjuntos de datos. Esta tecnología ha impulsado avances notables en dominios como el reconocimiento de voz y de imágenes, logrando niveles de precisión que en algunos casos igualan o superan las capacidades humanas (LeCun, Y., Bengio, Y., Hinton, 2015). Su principal fortaleza radica en su capacidad para adquirir representaciones complejas de los datos sin requerir de un diseño específico para cada tarea, lo cual le confiere gran

versatilidad y potencia dentro del ámbito de la inteligencia artificial (Goodfellow lan, Bengio Yoshua, 2016).

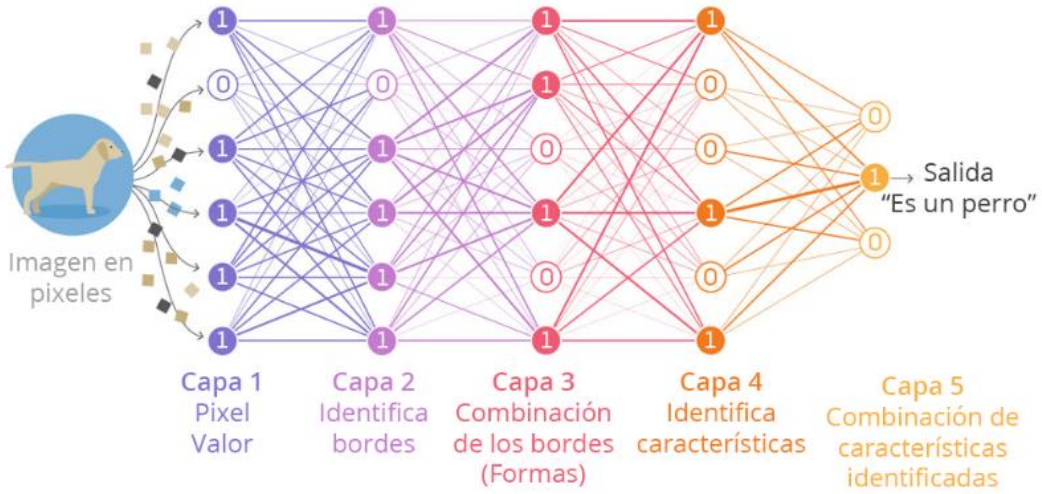


Figura 3. Esquema de reconocimiento de la imagen de un perro (Wolchover, 2020)

2.2.3. Historia y evolución

La evolución en inteligencia artificial (IA) y aprendizaje automático (AA) ha marcado un hito en el procesamiento del lenguaje natural (PLN). Con la llegada de modelos generativos como GPT (Generative Pre-trained Transformer) y BERT (Bidirectional Encoder Representations from Transformers), la habilidad de las máquinas para interpretar y producir lenguaje humano ha mejorado notablemente, permitiendo el desarrollo de aplicaciones como la traducción automática y los asistentes virtuales (Devlin et al., 2019; Vaswani et al., 2017). Estos modelos, impulsados por técnicas de aprendizaje profundo, aprovechan grandes volúmenes de datos para captar la complejidad lingüística, ofreciendo un rendimiento óptimo en la interpretación de textos (Goodfellow lan, Bengio Yoshua, 2016). A pesar de los retos éticos y de privacidad relacionados, su impacto positivo en el PLN continúa expandiéndose, apuntando a desarrollos futuros que harán estas aplicaciones más accesibles y avanzadas (T. B. Brown et al., 2020).

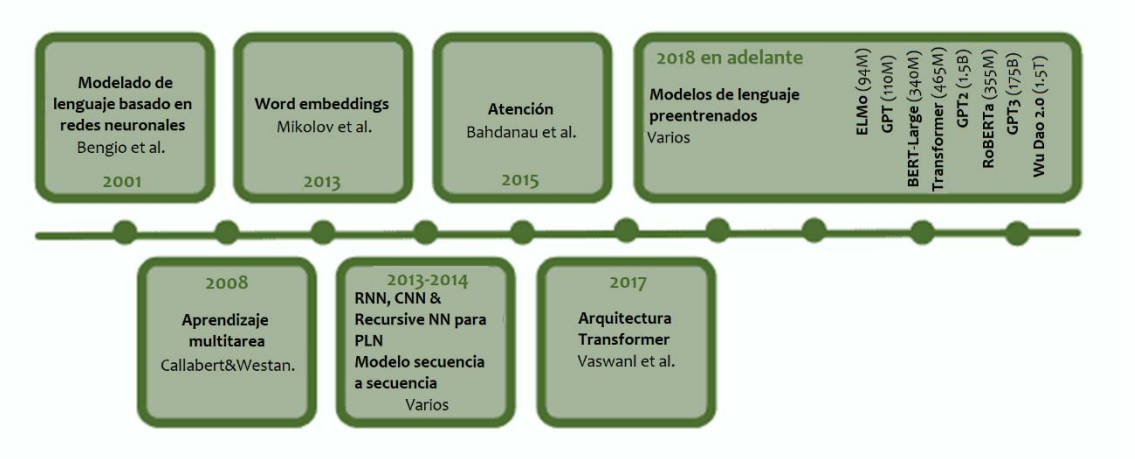


Figura 4. Historia contemporánea del PLN **(Diaz & Diaz, 2023)**

2.3. Procesamiento del Lenguaje Natural (PLN)

El Procesamiento del Lenguaje Natural (PLN) se dedica a mejorar la comunicación entre humanos y computadoras mediante el lenguaje natural. Este campo de la inteligencia artificial persigue que las máquinas sean aptas para entender, interpretar y producir lenguaje humano de forma significativa. Para lograr esto, se emplean técnicas y herramientas avanzadas que abarcan tareas como la traducción automática, el reconocimiento de voz y la generación de texto, entre otras. Con los progresos recientes en aprendizaje automático y redes neuronales, el PLN ha alcanzado un nivel de sofisticación que permite a las computadoras entender y reaccionar al lenguaje humano con mayor precisión y profundidad (Jurafsky, D., & Martin, J. H., 2009; Manning, C. D., & Schütze, H., 2000).

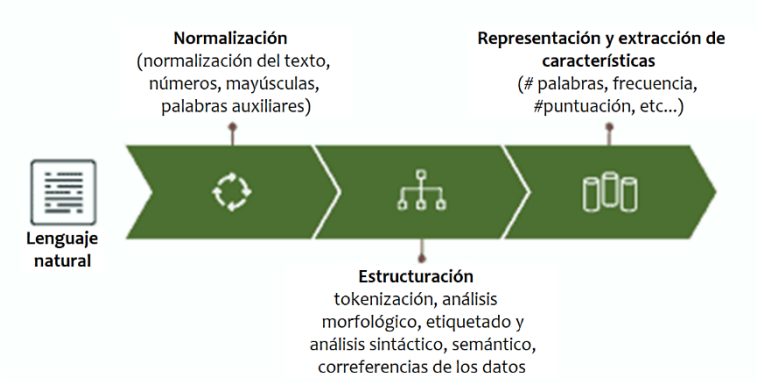


Figura 5. Proceso de preparación del texto recibe el nombre de construcción del corpus ***(Diaz & Diaz, 2023)***

2.3.1. Técnicas y modelos

El Procesamiento del Lenguaje Natural (PLN) ha avanzado notablemente con la introducción de modelos como las Redes Neuronales Recurrentes (RNN), las Redes de Memoria a Largo Plazo (LSTM) y, más recientemente, los modelos Transformer. Estas técnicas son fundamentales para interpretar y producir lenguaje natural, ya que permiten procesar secuencias complejas y aprender patrones de lenguaje a través del tiempo. Mientras que los RNN y LSTM han sido cruciales en tareas como la traducción automática y el reconocimiento de voz, los modelos Transformer han revolucionado el campo mediante el uso de mecanismos de atención, que optimizan la interpretación y creación del lenguaje al enfocarse en partes específicas del texto. Esto ha llevado a desarrollos como GPT y BERT, que establecen nuevos estándares en aplicaciones de PLN (Devlin et al., 2019; Vaswani et al., 2017).

Modelo	Descripción	Aplicaciones
RNN	Redes neuronales recurrentes	Traducción automática, reconocimiento de voz, generación de texto
LSTM	Redes neuronales de memoria a largo plazo	Comprensión del lenguaje natural, generación de código
Transformer	Arquitectura de red neuronal	Traducción automática, generación de texto, respuesta a preguntas

Figura 6. Modelos de Procesamiento de Lenguaje Natural (Elaboración propia)

2.3.2. Aplicaciones de PLN en la generación de código y documentación

El uso del Procesamiento del Lenguaje Natural (PLN) en la conversión de descripciones en lenguaje cotidiano a código fuente ha supuesto un avance notable en la manera en que los seres humanos se comunican con las computadoras. Herramientas como Codex de OpenAI han demostrado la viabilidad de interpretar instrucciones verbales y transformarlas en código funcional, lo cual impulsa la automatización y simplifica el desarrollo de software, haciéndolo accesible incluso a personas sin conocimientos de programación. Además, las aplicaciones de PLN han mejorado la generación de documentación de software al automatizar la creación de textos técnicos a partir del código fuente. Esta automatización facilita la comprensión y calidad de la documentación, un aspecto esencial en la eficiencia de los proyectos de software (Zaremba et al., 2021).

Figura 7. Copiloto de GitHub que utiliza OpenAI Codex para codificación (GitHub)

2.4. OpenAI y sus Modelos Generativos

OpenAI, fundada en 2015, es una organización de investigación que se enfoca en el desarrollo seguro y responsable de la inteligencia artificial (IA) para beneficio global. Su misión es asegurar que la IA de nivel general (AGI), con habilidades comparables a las humanas, sea desarrollada y utilizada en forma equitativa en todo el mundo. OpenAI ha liderado innovaciones significativas en modelos generativos, destacándose con su serie de modelos GPT (Generative Pre-trained Transformer). Estos modelos, desde su primera versión hasta las más recientes iteraciones, como GPT-4, han establecido estándares en el procesamiento del lenguaje natural (PLN) debido a su habilidad de generar textos coherentes, traducir idiomas y comprender el contexto en la lectura, representando un avance considerable en la generación y comprensión de lenguaje humano (T. B. Brown et al., 2020; OpenAI, n.d.; Radford Alec et al., 2019).

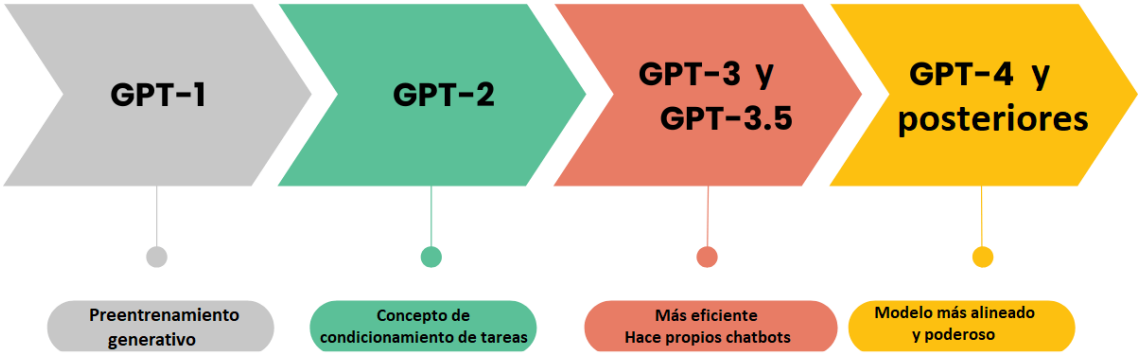


Figura 8. Evolución de la serie Chat GPT (Hussain, 2023)

2.4.1. Metodologías de entrenamiento

Para desarrollar sus modelos avanzados de inteligencia artificial, OpenAI emplea metodologías de entrenamiento innovadoras, en particular para la serie de modelos Generative Pretrained Transformer (GPT). El proceso comienza con una fase de preentrenamiento, donde el modelo se capacita utilizando un extenso grupo de datos que abarca textos de diversa índole, permitiéndole adquirir una comprensión básica de la estructura y gramática del lenguaje. Esta base inicial facilita al modelo el aprendizaje de contextos amplios y variados (Radford Alec et al., 2019).

Posteriormente, se realiza un ajuste fino (fine-tuning), que se compone de adaptar el modelo a tareas específicas de procesamiento de lenguaje natural (PLN) mediante un grupo de datos centrado en la tarea deseada, permitiendo una mayor precisión y especialización en áreas concretas. (T. B. Brown et al., 2020).

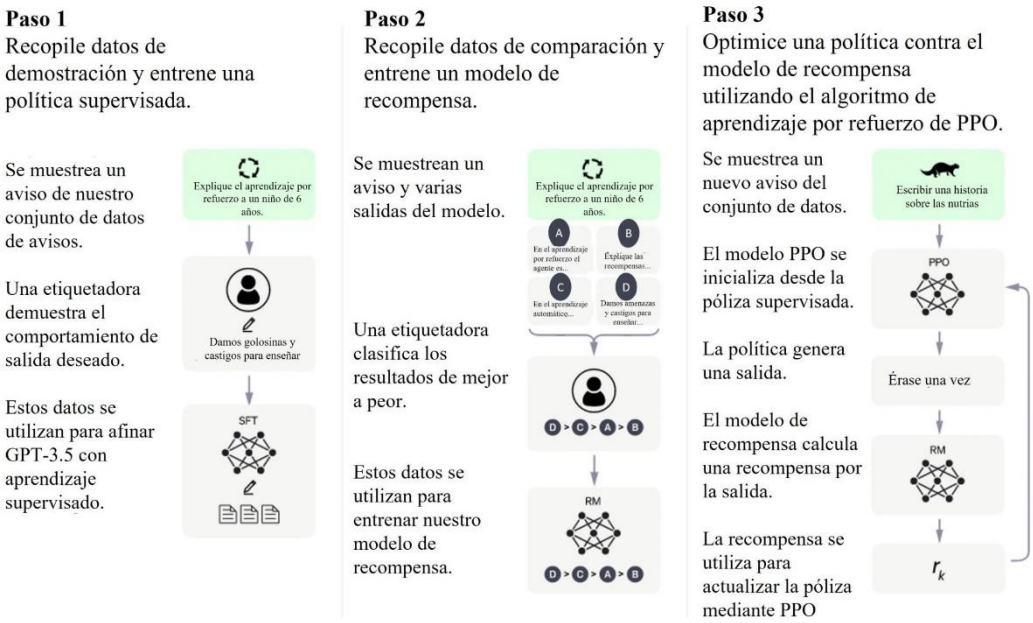


Figura 9. Modalidad de entrenamiento de ChatGPT de OpenAI (OpenAI, n.d.)

2.4.2. Fine-Tuning (Ajuste Fino) en Modelos de Lenguaje Natural

El proceso de ajuste fino o *fine-tuning* en modelos de lenguaje, como los desarrollados por OpenAI, permite especializar un modelo preentrenado para que cumpla con tareas específicas. Este procedimiento se realiza entrenando el modelo con un conjunto de datos especializado, optimizado para el contexto en el cual se desea ganar rendimiento del modelo. Al aplicar el ajuste fino, el modelo es capaz de adaptarse a tareas concretas sin necesidad de un nuevo entrenamiento desde cero, logrando una mayor precisión y eficacia en aplicaciones específicas como la clasificación de textos, la generación de contenido estructurado, o el funcionamiento de chatbots.

El fine-tuning es útil en aplicaciones donde la información de entrenamiento es limitada, pero se requiere un elevado nivel de precisión en tareas específicas. El modelo preentrenado ya ha sido expuesto a volúmenes significativos de datos generales, lo que le permite aprovechar conocimientos previos, mientras que el ajuste fino afina esos conocimientos a los requerimientos particulares. Este enfoque

se ha demostrado eficiente, reduciendo los costos computacionales y acelerando el proceso de entrenamiento (Brown et al., 2020).

Durante el proceso, es importante ajustar la tasa de aprendizaje y el número de épocas para evitar el sobreajuste y mantener un equilibrio entre la generalización y la especialización del modelo. Además, el uso de técnicas como la regularización y la validación cruzada es común para asegurar la robustez del modelo ajustado (Zhang et al., 2021).

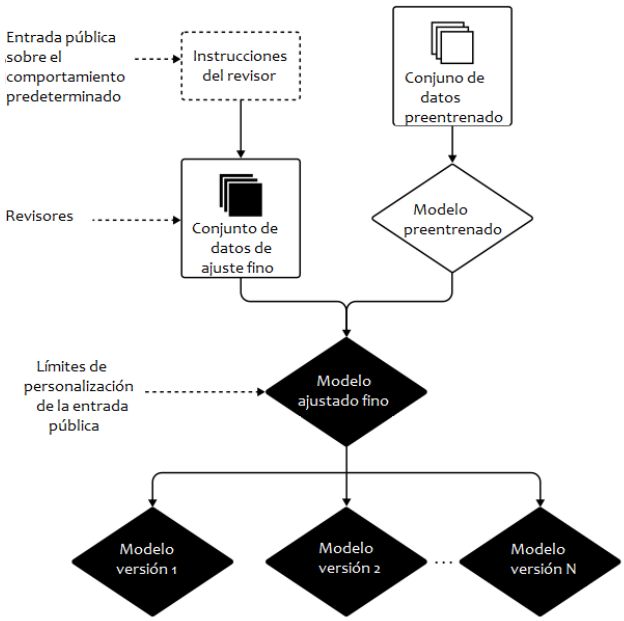


Figura 10. Diagrama de funcionamiento del Modelo Afinado (Fined-tuned model) ***(OpenAI, n.d.)***

2.5. Lenguaje de Modelado Unificado (UML)

El Lenguaje de Modelado Unificado (UML) es una herramienta estándar en la ingeniería de software, proporcionando un marco que facilita la representación visual de sistemas complejos en términos de diseño, implementación y documentación. Desde su desarrollo por el Object Management Group (OMG), UML se ha consolidado como un lenguaje de modelado ampliamente aceptado en la industria, permitiendo que profesionales de todo el mundo comuniquen ideas técnicas de manera uniforme y comprensible (Fuentes & Vallecillo, 2004). UML incluye una variedad de diagramas, como los de casos de uso, que detallan las relaciones entre el sistema y usuarios, los de clases, que estructuran las entidades del sistema y sus relaciones, y los de secuencia, que ilustran la dinámica temporal entre componentes.

La adaptabilidad de UML a diferentes metodologías de desarrollo, como la programación ágil y el desarrollo dirigido por modelos, refuerza su utilidad en múltiples contextos. Su estandarización facilita también la colaboración interdisciplinaria, proporcionando un lenguaje común que supera barreras de especialización. Sin embargo, la creación manual de diagramas UML puede ser laboriosa y propensa a errores, lo cual motiva la exploración de soluciones automatizadas que aprovechen los progresos en inteligencia artificial y procesamiento del lenguaje natural para mejorar la creación de diagramas.

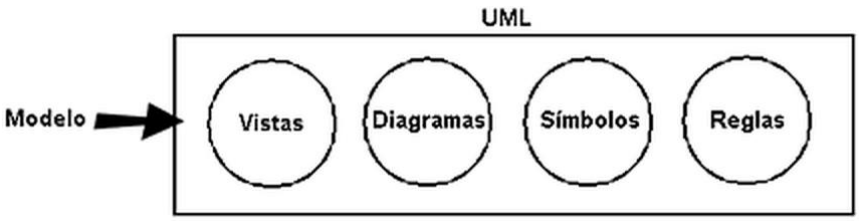


Figura 11. Modelo UML (Elaboración propia)

2.5.1. PlantUML

PlantUML es una herramienta fundamental en la ingeniería de software para la elaboración rápida y programática de diagramas UML. Su sintaxis, sencilla y basada en texto, permite a los desarrolladores y diseñadores documentar eficazmente estructuras complejas de software y flujos de procesos sin depender de interfaces gráficas extensas. Con soporte para una variedad de tipos de diagramas, como los diagramas de secuencia, de casos de uso, de clases, de actividades, de componentes y de estados; PlantUML facilita tanto la documentación como el diseño de sistemas complejos, lo que la hace una alternativa versátil dentro del desarrollo de software (S. Brown, 2020). Comparado con otras herramientas de diagramación UML, PlantUML destaca por su enfoque en la simplicidad y la automatización. Mientras que aplicaciones gráficas como Microsoft Visio o Lucidchart brindan interfaces visuales ricas, PlantUML permite la creación rápida de diagramas a partir de descripciones textuales, una característica especialmente útil en entornos de desarrollo ágil, donde la documentación debe mantenerse actualizada a medida que evolucionan los requisitos del software. Además, su capacidad de generar diagramas UML de manera automatizada la convierte en una herramienta ideal para la integración continua y la documentación automatizada, superando en algunos aspectos a otras herramientas que pueden requerir pasos adicionales o complementos específicos (Ambler, 2002).

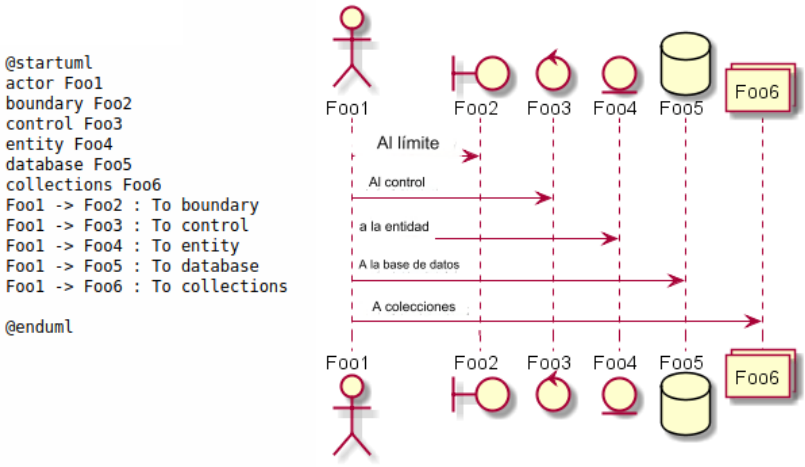


Figura 12. Código y Diagrama PlantUML (PlantUML Website, *n.d.)*

2.5.2. Generación de Diagramas UML a través de PlantUML

Los diagramas UML proporcionan una representación visual estandarizada para entender y expresar la estructura y el comportamiento de sistemas complejos en el campo de la ingeniería de software. Incluyen tanto diagramas estructurales, como los de clases y objetos que representan la arquitectura y las relaciones, como diagramas de comportamiento, como los de secuencia y casos de uso, que describen la interacción dinámica entre los componentes del sistema bajo diversas condiciones (Grady et al., 2005). Esta diversidad permite que desarrolladores y diseñadores capten y documenten los detalles técnicos del software, lo cual resulta esencial para el análisis, el diseño, la implementación y el mantenimiento de sistemas.

PlantUML facilita la generación de estos diagramas UML mediante un lenguaje de marcado sencillo. Los usuarios pueden describir los elementos y sus relaciones en texto, y PlantUML convierte estas descripciones en diagramas visuales en varios formatos, como PNG y SVG, así como en formato ASCII para uso en documentos de texto (Brown, 2019).

En esta tesis, PlantUML actúa como la herramienta principal para transformar descripciones en lenguaje cotidiano, interpretadas por el modelo de OpenAI, en diagramas UML, contribuyendo a la automatización y sincronización constante de la documentación del diseño con el código fuente y la documentación técnica.

La elección de PlantUML se justifica no solo por su capacidad de integración con otros sistemas a través de APIs, sino también por su flexibilidad y soporte para personalizaciones avanzadas. PlantUML puede ser integrado en diversos entornos de desarrollo, como IDEs populares, plataformas para gestionar versiones, y pipelines de CI/CD, lo que lo hace una opción óptima para la automatización en el desarrollo de software (Schindler & Schäfer, 2021).

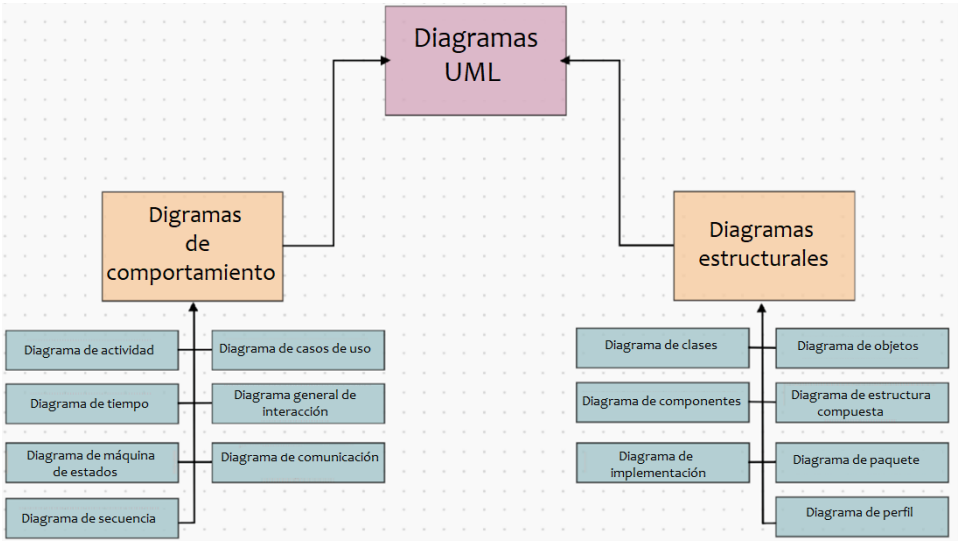


Figura 13. Diagramas UML (Elaboración propia)

2.6. APIs y Microservicios

En el contexto de creación de software moderno, las APIs (Interfaz de Programación de Aplicaciones) y los microservicios son esenciales en la arquitectura y funcionamiento de sistemas distribuidos. Las APIs posibilitan la comunicación entre varios componentes de software, promoviendo la interoperabilidad y la integración de servicios. Además, los microservicios representan un estilo arquitectónico que organiza una aplicación en múltiples servicios pequeños, independientes y que

pueden desplegarse de forma autónoma. Esta combinación de APIs y microservicios ha transformado la forma en que las aplicaciones escalables y mantenibles se diseñan, implementan y despliegan (Richardson, 2018).

El uso de microservicios y APIs es particularmente relevante en sistemas que requieren flexibilidad y escalabilidad, como en la implementación de un modelo de IA que genera diagramas UML en esta tesis. Los microservicios permiten descomponer la funcionalidad del sistema en módulos específicos. Las APIs actúan como un contrato que define cómo estos microservicios se comunican entre sí y con otros sistemas, facilitando la integración y la extensión del sistema en el futuro (Newman, 2021).

2.6.1. Diseño e Implementación de APIs

La creación y desarrollo de APIs es un proceso clave que determina cómo se relacionan los distintos elementos dentro de un sistema de software. Un buen diseño de API debe ser intuitivo, fácil de usar, y suficientemente robusto para manejar errores y excepciones. Las APIs RESTful, que utilizan el protocolo HTTP, se han consolidado como el estándar predominante en la industria gracias a su simplicidad, escalabilidad y alineación con los principios de la web. Estas APIs facilitan que los clientes lleven a cabo operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los recursos del sistema de modo estandarizado y seguro (Fielding, 2000).

En el caso de esta tesis, la API diseñada es responsable de gestionar la comunicación entre el modelo de OpenAI y PlantUML, así como con el chatbot de Telegram. La API debe ser capaz de recibir entradas en texto natural, procesarlas mediante el modelo de IA, y devolver el diagrama UML generado. Además, debe manejar la verificación de identidad, el control de acceso y la administración de sesiones de los usuarios que interactúan con el sistema a través del chatbot de Telegram. La aplicación de prácticas recomendadas en el diseño de APIs, como la adherencia a los principios RESTful, el uso de HTTP status codes, y la documentación clara y

detallada, son esenciales para garantizar la fiabilidad y mantenibilidad del sistema (Massé, 2011).

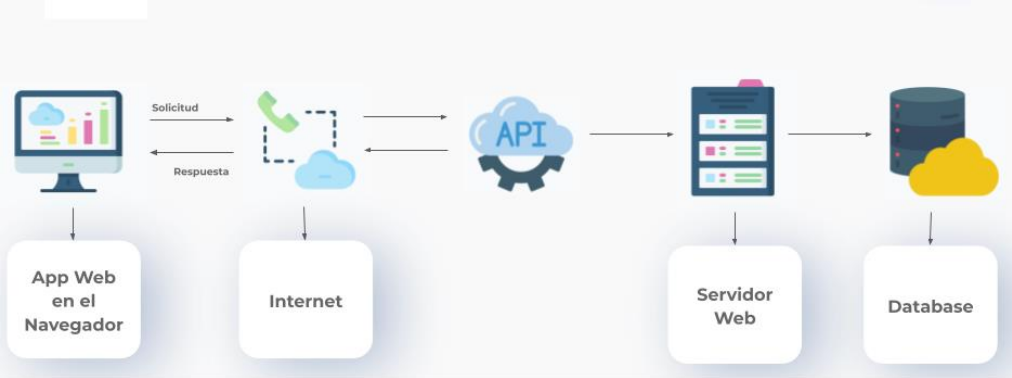


Figura 14. Flujo de petición de información a través de una API ***(Aguilera, 2024)***

2.6.2. Integración de Microservicios

La integración de microservicios en un sistema de software implica coordinar múltiples servicios autónomos para trabajar juntos de manera coherente. Dado que cada microservicio opera de manera autónoma, la integración se realiza a través de APIs, que facilitan la interacción entre los servicios. Una de las principales ventajas de los microservicios es su capacidad para desarrollarse, desplegarse y escalarse de forma independiente, lo cual permite implementar cambios y agregar nuevas funcionalidades sin interferir con otros componentes del sistema (Lewis & Fowler, 2014).

En el contexto de esta tesis, el enfoque de arquitectura de microservicios permite la separación de responsabilidades, donde cada microservicio maneja una parte específica del proceso: la percepción del lenguaje natural, la creación de diagramas UMLy la interacción con los usuarios a través del chatbot de Telegram. La integración se realiza utilizando una arquitectura basada en eventos, donde los microservicios se comunican entre sí mediante mensajes asíncronos o solicitudes HTTP, asegurando la consistencia y la disponibilidad del sistema. Esta arquitectura modular facilita la escalabilidad y el mantenimiento del sistema, permitiendo que los diferentes equipos de desarrollo trabajen de manera paralela en distintos microservicios sin interferencias (Newman, 2021).

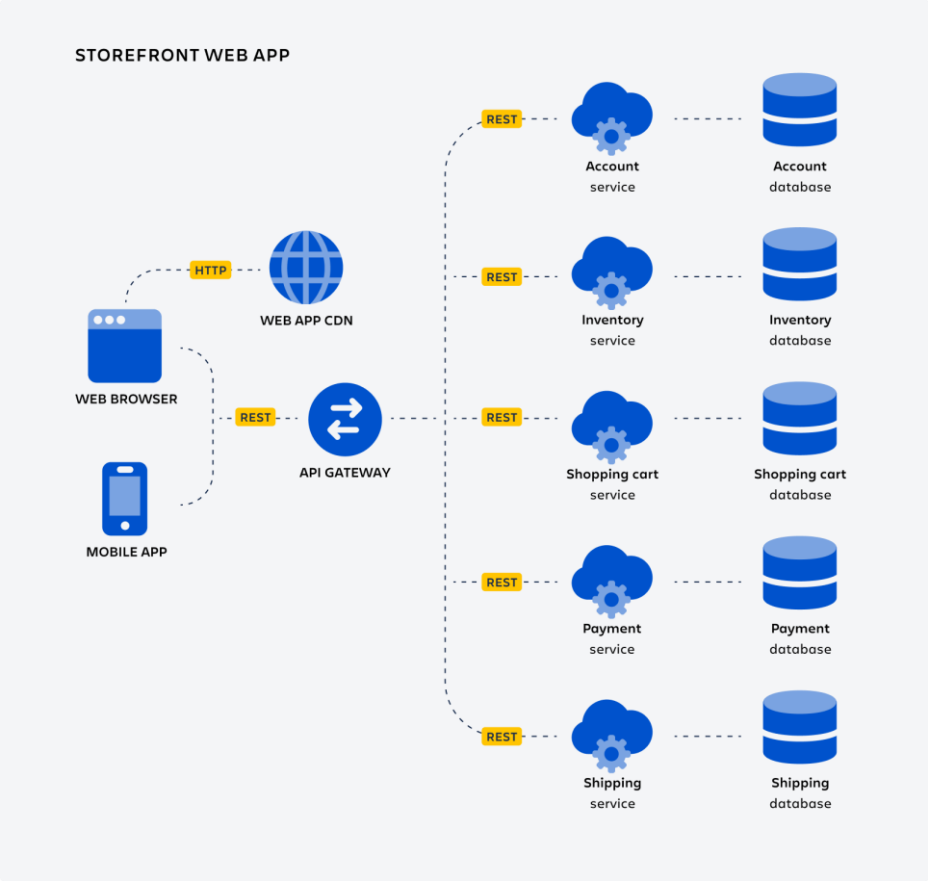


Figura 15. Aplicación web de una tienda con microservicios (Atlassian, n.d.)

2.7. Integración de Chatbots en Entornos de Desarrollo

La integración de chatbots en entornos de creación de software ha adquirido relevancia en los últimos años por su capacidad para simplificar la interacción entre los usuarios y los sistemas de desarrollo. Los chatbots, al aprovechar técnicas avanzadas de procesamiento del lenguaje natural (PLN) y aprendizaje automático, logran interpretar comandos en lenguaje natural y realizar tareas automatizadas que antes requerían intervención manual. Esta automatización no solo incrementa

la eficacia, sino que también reduce la posibilidad de errores humanos en actividades repetitivas o complejas (Joulin, et al., 2017).

La incorporación de chatbots en el ciclo de vida del desarrollo de software ofrece múltiples beneficios, como la agilización de procesos, la mejora en la interacción entre los equipos de desarrollo y la capacidad de proporcionar asistencia inmediata a los desarrolladores. Asimismo, los chatbots pueden configurarse para llevar a cabo funciones específicas, como la generación de código, la gestión de tareas y la elaboración de diagramas UML, entre otras. Adaptar estas herramientas a los requerimientos particulares de cada proyecto permite mejorar el rendimiento y aumentar la satisfacción del equipo (Ramesh, 2020).

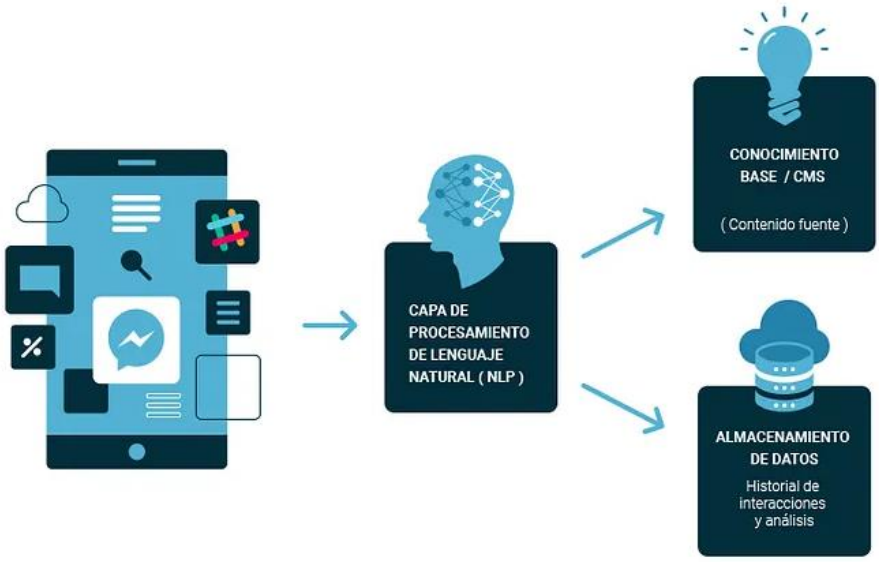


*Figura 16. Arquitectura de un chatbot (*Comprendiendo La Arquitectura de Los Chatbots*, n.d.)*

2.7.1. Funcionalidad y Usabilidad del Chatbot de Telegram

Telegram es una de las plataformas de mensajería más reconocidas a nivel mundial, y su API abierta facilita la creación de chatbots con múltiples funcionalidades. Un chatbot en Telegram puede ser programado para interactuar con los usuarios

mediante comandos simples de texto, facilitando el acceso a funcionalidades complejas sin necesidad de un conocimiento técnico profundo (Kraus & Feuerriegel, 2017). En el área del desarrollo de software, un chatbot de Telegram consigue actuar como un intermediario entre el usuario y las herramientas de desarrollo, permitiendo la creación y modificación de diagramas UML directamente desde una conversación.

La usabilidad del chatbot de Telegram es clave para su adopción en entornos de desarrollo. Debe ser intuitivo, rápido y capaz de manejar diferentes formatos de entrada del usuario. La capacidad de respuesta instantánea y la facilidad de uso son factores críticos que determinan su efectividad. El chatbot, además, debe ser suficientemente flexible para integrarse con otras herramientas y APIs empleadas en el desarrollo de software, asegurando una interacción de usuario continua y sin interrupciones. (Vanichvasin, 2016).

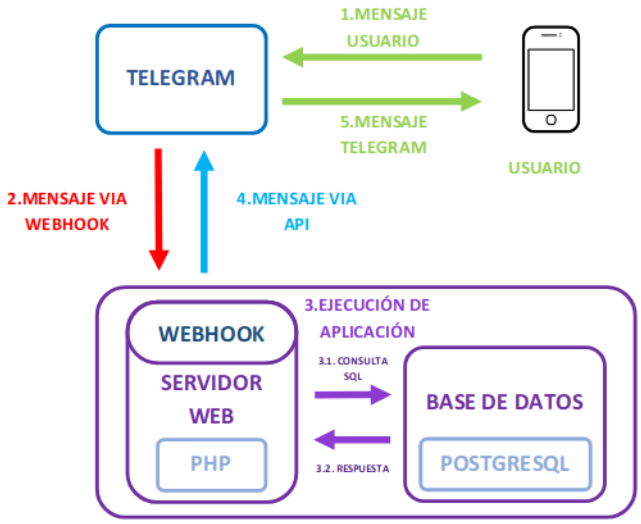


Figura 17. Flujo de información de chatbot con Telegram ***(Álvaro, 2017)***

2.7.2. Interacción en Tiempo Real para la Generación de Diagramas UML

La creación de diagramas UML mediante un chatbot en Telegram constituye un avance importante en la automatización de tareas dentro del desarrollo de

software. Mediante el uso de descripciones textuales, los usuarios solicitan la creación de diagramas UML, que son generados y devueltos en tiempo real. Este enfoque no solo ahorra tiempo, sino que también permite a los desarrolladores concentrarse en tareas de mayor importancia, dejando la implantación de diagramas a una herramienta automatizada (Chen, et al., 2021).

La interacción en tiempo real es un aspecto crucial de esta funcionalidad. El chatbot debe ser apto para procesar las solicitudes de los usuarios rápidamente y entregar retroalimentación inmediata. Esto es posible gracias a la integración de tecnologías como OpenAI y PlantUML, que interpretan las descripciones de texto y las convierte en representaciones visuales precisas (Francescucci & Rohani, 2019). Asimismo, la cualidad de gestionar cambios y actualizaciones de manera inmediata garantiza que los diagramas generados reflejen con precisión los requisitos actuales del sistema, lo que mejora la calidad y coherencia del diseño del software.

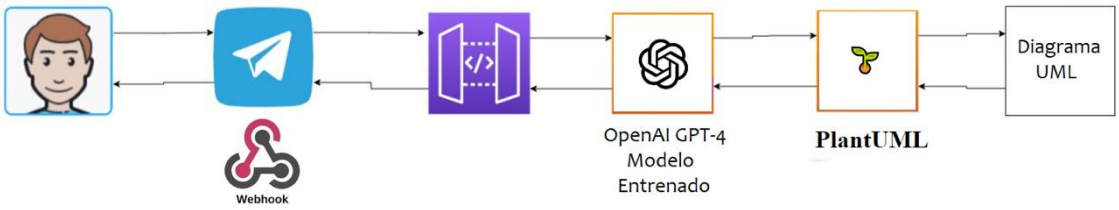


Figura 18. Integración APIs con Telegram ***(Avuthu, 2023)***

2.8. Ciclo de Vida del Desarrollo de Software

El ciclo de vida del desarrollo de software (SDLC, por sus siglas en inglés, Systems Development Life Cycle) es un sistema organizado que establece las etapas esenciales para desarrollar sistemas de software de gran calidad. Este ciclo abarca fases como planificación, análisis, diseño, implementación, pruebas, despliegue y mantenimiento, cada una desempeñando un papel fundamental para asegurar que el software final satisface los requisitos establecidos y no tenga defectos importantes (Pressman, 2014). El SDLC ofrece un enfoque sistemático para

enfrentar los desafíos del desarrollo de software, y su adecuada aplicación es vital para el resultado exitoso de los proyectos en este campo.

La introducción de nuevas tecnologías, como la automatización basada en inteligencia artificial, está transformando cómo se llevan a cabo estas etapas. La automatización agiliza muchas de las tareas más repetitivas y propensas a errores, desde la generación de código hasta la creación de documentación, lo que permite a los equipos de desarrollo concentrarse en aspectos más estratégicos del proyecto (Bass, Clements, & Kazman, 2012).

Figura 19. Ciclo de vida de desarrollo de aplicaciones (TRPlane, 2021)

2.8.1. Impacto de la Automatización en el Ciclo de Vida

La automatización en el SDLC ha transformado profundamente el desarrollo de proyectos de software. Al automatizar tareas esenciales como la creación de pruebas, la integración continua y la generación de documentación, se incrementa la eficiencia y consistencia en cada periodo del ciclo de vida del desarrollo. Según Sommerville (2015), la automatización no solo disminuye el tiempo y el esfuerzo requeridos para completar estas tareas, sino que también minimiza la probabilidad de fallos humanos, lo que contribuye a un software de mayor calidad y a un proceso de desarrollo más predecible.

En particular, la automatización en las etapas tempranas del SDLC, como la planificación y el diseño, lleva a una mayor precisión en la captura de requisitos y en la creación de modelos de software, lo que a su vez reduce la necesidad de retrabajo en fases posteriores. Este enfoque contribuye a mejorar la productividad general del equipo de desarrollo y a acortar los plazos de entrega (Westland, 2002). Además, la automatización ofrece una mayor flexibilidad ante los cambios, dado que facilita implementar modificaciones en los sistemas de forma más rápida y efectiva, sin comprometer la calidad del producto final (Fowler, 2004).

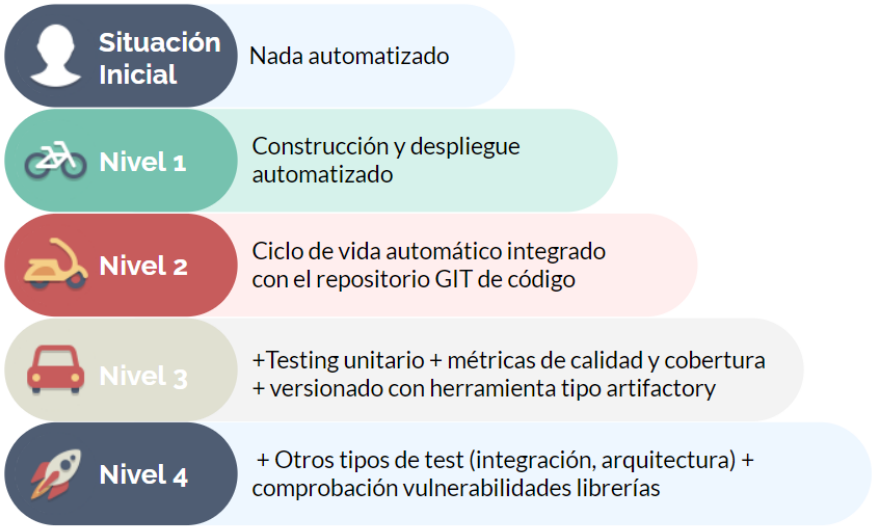


Figura 20. Automatización del ciclo de vida software ***(Admin, 2023)***

2.8.2. Optimización de Tiempos de Diseño

La etapa de diseño en el SDLC es fundamental, pues establece la arquitectura y estructura que orientarán el desarrollo del software. No obstante, esta fase puede resultar una de las más extensas, especialmente en proyectos de gran envergadura y complejidad. La optimización de los tiempos de diseño mediante herramientas automatizadas y modelos de inteligencia artificial ha demostrado ser una estrategia eficaz para incrementar la eficiencia en esta fase.

En el informe "State of DevOps" de Google Cloud destaca las métricas de rendimiento esenciales para la entrega de software: frecuencia de despliegue,

tiempo de entrega de cambios, tiempo de restauración del servicio y tasa de fallos de cambios. Estas métricas, conocidas como métricas DORA, son fundamentales para analizar la agilidad y solidez de los equipos de desarrollo. El informe revela que los equipos de alto rendimiento logran implementar cambios de forma más rápida y con menos errores, mejorando así su habilidad para adaptarse a las demandas del mercado y reduciendo el tiempo de inactividad (*Accelerate State of DevOps 2021 | Google Cloud*, n.d.).

Métrica de rendimiento de entrega de software	Élite	Alto	Medio	Bajo
Frecuencia de despliegue Para la aplicación o servicio principal en el que trabaja, ¿con qué frecuencia su organización implementa código en producción o lo libera para los usuarios finales?	Bajo demanda (múltiples implementaciones por día)	Entre una vez por semana y una vez por mes	Entre una vez al mes y una vez cada 6 meses	Menos de una vez cada seis meses
Plazo de ejecución de los cambios Para la aplicación o servicio principal en el que trabaja, ¿cuál es el plazo de entrega de los cambios (es decir, cuánto tiempo lleva pasar del código confirmado al código que se ejecuta correctamente en producción)?	Menos de una hora	Entre un día y una semana	Entre un mes y seis meses	Más de seis meses
Es hora de restablecer el servicio Para la aplicación o servicio principal en el que trabaja, ¿cuánto tiempo suele tardar en restablecerse el servicio cuando ocurre un incidente o un defecto que afecta a los usuarios (por ejemplo, una interrupción no planificada o un deterioro del servicio)?	Menos de una hora	Menos de un día	Entre un día y una semana	Más de seis meses
Cambiar la tasa de fallos Para la aplicación o servicio principal en el que trabaja, ¿qué porcentaje de cambios en producción o lanzados a los usuarios resultan en un servicio degradado (por ejemplo, conducen a un deterioro o interrupción del servicio) y posteriormente requieren una solución (por ejemplo, requieren una revisión, una reversión, una corrección posterior, un parche)?	0%-15%	16%-30%	16%-30%	16%-30%

Figura 21. Métrica de rendimiento de entrega de software (Accelerate State of DevOps 2021 | Google Cloud*, n.d.)*

Además, la capacidad de realizar iteraciones rápidas en los diseños facilita que los equipos de desarrollo se adapten rápidamente a los cambios en los requisitos, lo cual es esencial en entornos de desarrollo dinámicos y en enfoques ágiles. La optimización de los tiempos de diseño, por lo tanto, no solo contribuye a reducir el tiempo total de desarrollo, sino que también mejora la calidad del software al permitir una mayor exploración y refinamiento de las soluciones antes de la fase de implementación (Beck, 1999).

Figura 22. Ciclo de la automatización de procesos ***(Belandria, 2023)***

2.9. Estudios de Caso y Aplicaciones Relevantes

El análisis de casos prácticos y aplicaciones significativas es esencial para comprender cómo la integración de modelos de inteligencia artificial y herramientas de automatización, como OpenAI y PlantUML, transforma el ciclo de vida del desarrollo de software. Estos estudios proporcionan una visión detallada de cómo las empresas y organizaciones han implementado tecnologías similares para resolver problemas complejos, aumentar la eficiencia del desarrollo y disminuir los plazos y gastos relacionados con la creación de software.

Uno de los estudios de caso más ilustrativos es el uso de modelos generativos de inteligencia artificial en la automatización del diseño de software. En este contexto, Google ha sido pionera en el desarrollo de soluciones de inteligencia artificial que posibilitan la creación automática de código y diseño de software a partir de texto natural, lo que ha llevado a una mejora significativa en la productividad de sus desarrolladores y una disminución en el tiempo requerido para implementar

nuevos productos al mercado (Google AI, 2018). La aplicación de estos modelos en entornos de desarrollo ágil ha permitido a los equipos responder rápidamente a los cambios en los requisitos, ajustando los diseños de manera dinámica sin sacrificar la calidad del software.

Otro ejemplo relevante es el caso de IBM, que ha integrado técnicas avanzadas de procesamiento de lenguaje natural y automatización en su plataforma Watson para facilitar la generación de diagramas UML y otros artefactos de diseño a partir de requisitos escritos con descripciones textuales. Este enfoque no solo ha optimizado los tiempos de diseño, sino que también ha mejorado la precisión de los modelos generados, reduciendo el número de errores y revisiones necesarias en las fases posteriores del ciclo de vida del software (IBM Watson, 2020). La experiencia de IBM demuestra cómo la combinación de inteligencia artificial y herramientas automatizadas logran transformar las prácticas tradicionales de ingeniería de software, permitiendo a las organizaciones innovar y adaptarse más rápidamente a las demandas del mercado.

Además, el estudio de empresas como Atlassian, que ha implementado herramientas de automatización en su suite de productos, muestra cómo la integración de tecnologías de inteligencia artificial en herramientas de gestión de proyectos y desarrollo de software consigue mejoras significativas en la colaboración y la eficiencia del equipo. Atlassian ha utilizado la automatización para simplificar tareas repetitivas, permitiendo a los desarrolladores centrarse en tareas más complejas y creativas, lo que ha resultado en un aumento en la productividad y una reducción en el tiempo de comercialización de sus productos (Atlassian, 2019).

Estos casos de estudio y aplicaciones evidencian que la adopción de tecnologías avanzadas de inteligencia artificial y automatización no solo es factible, sino que también proporciona beneficios concretos en cuanto a eficiencia, calidad y rapidez en el desarrollo de software. La integración de estas tecnologías, como se plantea en esta investigación, tiene el potencial de transformar profundamente la manera

en que se diseñan y desarrollan sistemas de software, acortando la distancia entre la conceptualización inicial y la implementación final.

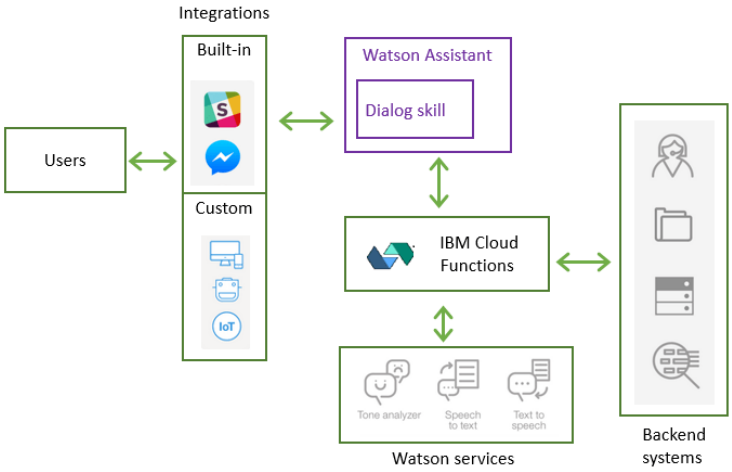


Figura 23. Integración de un chatbot con el asistente IBM Watson (IBM Watson Assistant, Telegram and Python, *n.d.)*

2.10. Retos y consideraciones

La implementación de un proceso iterativo de entrenamiento para un modelo de OpenAI, integrado con PlantUML para generar diagramas, enfrenta varios retos y aspectos a considerar para garantizar su éxito y aplicación en entornos reales de desarrollo de software. Estos se pueden clasificar en desafíos técnicos, operativos y de interacción humano-computadora.

- Retos Técnicos:

Uno de los principales desafíos es la precisión en el procesamiento del lenguaje natural (PLN). Aunque los modelos de OpenAI han logrado avances notables en la comprensión y generación de lenguaje natural, la traducción de descripciones textuales a diagramas UML precisos continúa siendo complicada, principalmente

debido a la ambigüedad inherente al lenguaje humano (Manning et al., 2014). Es fundamental que los modelos comprendan los contextos específicos del dominio del software, lo cual exige un entrenamiento exhaustivo y de calidad para reducir errores de interpretación.

La integración de PlantUML con OpenAI añade un reto adicional relacionado con la consistencia y coherencia de los diagramas generados. Es crucial que los diagramas UML no solo sean correctos desde el punto de vista sintáctico, sino que también sean apropiados semánticamente en el contexto de cada proyecto de software. La falta de coherencia podría llevar a errores en la interpretación y, finalmente, a fallas en el diseño del software (OMG, 2017).

- Retos Operativos:

Desde el punto de vista operativo, es necesaria una infraestructura robusta y escalable para entrenar de forma continua un modelo que se ajuste a nuevas descripciones y datos. El entrenamiento iterativo de modelos de IA puede ser exigente en cuanto a recursos computacionales y tiempo, especialmente en entornos de desarrollo dinámicos (LeCun et al., 2015). Las organizaciones deben evaluar estos costos frente a los beneficios que podría ofrecer la automatización para determinar la viabilidad del proyecto.

Además, mantener actualizados tanto el modelo de OpenAI como su integración con PlantUML para reflejar cambios en los requisitos del software es un reto continuo. A medida que los modelos y herramientas evolucionan, el proceso de elaboración de diagramas UML requiere ajustes frecuentes para asegurar la precisión y la relevancia del sistema.

- Retos de Interacción Humano-Computadora:

La interacción con los usuarios, a través del chatbot de Telegram integrado, plantea otro reto importante. Hay que asegurar que el chatbot interprete correctamente

las instrucciones de los usuarios y genere diagramas UML precisos es fundamental para la usabilidad del sistema. Las dificultades en la comprensión del contexto o la ambigüedad en las instrucciones de los usuarios pueden resultar en respuestas incorrectas, afectando la confiabilidad y la aceptación del sistema (Shneiderman, 2020).

- Enfoques y Consideraciones de la Investigación:

Además de los desafíos mencionados, se identifican aspectos inherentes al enfoque de la investigación. La dependencia de los modelos de OpenAI implica que cualquier restricción presente en estos modelos, tales como los sesgos en los datos de entrenamiento o la capacidad de generalización, influirá en los resultados del sistema implementado (Bender et al., 2021). Asimismo, la integración con PlantUML, aunque potente, podría no ser lo suficientemente flexible para manejar todos los casos de uso en entornos de desarrollo complejos, lo que reduce la aplicabilidad universal de la solución propuesta.

Finalmente, la evaluación del impacto de este sistema en la optimización del ciclo de vida del software y en la interacción con los usuarios está circunscrita a los entornos y contextos específicos en los que se desarrolla la investigación. La extrapolación de los resultados a otros contextos puede requerir estudios adicionales y ajustes del modelo.

2.11. Relación entre el marco teórico y la metodología

El marco teórico presentado en este trabajo sirve como base fundamental para las decisiones metodológicas tomadas en el desarrollo de la investigación. Específicamente:

- **Procesamiento del Lenguaje Natural (PLN):** Los conceptos relativos al PLN permiten comprender cómo el modelo de OpenAI puede interpretar descripciones en lenguaje natural y transformarlas en comandos específicos

para generar diagramas UML. Esto establece la capacidad del sistema para interactuar de manera efectiva con los usuarios mediante un lenguaje accesible.

- **PlantUML:** La elección de esta herramienta se justifica en su capacidad para convertir texto estructurado en diagramas visuales de manera automatizada, alineándose con los objetivos de accesibilidad y eficiencia del proyecto. Esta herramienta también facilita la integración en entornos de desarrollo continuo gracias a su compatibilidad con APIs.

- **Ajuste fino (fine-tuning):** El enfoque iterativo de ajuste fino (fine-tuning) garantiza que el modelo se adapte de manera precisa a las descripciones específicas del dominio del software. Esto permite que el sistema genere diagramas UML que reflejen con exactitud las necesidades de diseño planteadas por los usuarios.

- **Microservicios y APIs:** Los principios de diseño modular, explicados en el marco teórico, permiten la integración eficiente del modelo de IA con PlantUML y el chatbot de Telegram, asegurando una comunicación fluida entre los componentes.

Estos conceptos proporcionan los cimientos para la metodología implementada en este trabajo, asegurando que las decisiones técnicas y metodológicas se fundamenten en una base sólida y coherente con los objetivos del proyecto.

3. Desarrollo de La Solución

El proceso de creación de diagramas UML es clave en el diseño de software, aunque puede resultar laborioso y susceptible a errores cuando se lleva a cabo manualmente. Esta tesis propone una solución automatizada mediante el entrenamiento de un modelo de OpenAI, integrado con PlantUML y un chatbot de Telegram, para generar diagramas UML a partir de descripciones en lenguaje natural.

La solución emplea las capacidades de procesamiento de lenguaje natural de OpenAI, la API de PlantUML y la interacción con usuarios a través de Telegram, implementada en un entorno de microservicios con Docker, esta herramienta busca optimizar los tiempos de diseño y mejorar la accesibilidad de las herramientas de modelado. A través de un proceso iterativo de entrenamiento y evaluación, se afina continuamente el modelo para incrementar la exactitud y la eficiencia en la generación de diagramas UML.

3.1. Metodología

La investigación propuesta para el desarrollo de esta solución sigue una metodología basada en un enfoque exploratorio y experimental, combinando la recolección y análisis de datos con un proceso iterativo de pruebas para ajustar el modelo de OpenAI en la creación de diagramas UML.

Tipo de Investigación

1. **Investigación Exploratoria:**

   Esta investigación se clasifica como exploratoria, ya que busca profundizar en el uso de modelos de lenguaje natural (NLP) de OpenAI, una tecnología emergente, en el contexto de la generación automática de diagramas UML. El propósito es investigar nuevas capacidades de los modelos de inteligencia

artificial para traducir descripciones textuales a diagramas UML, optimizando el ciclo de vida de desarrollo de software. De acuerdo con Stebbins (2001), la investigación exploratoria se utiliza cuando el objetivo es "ganar una comprensión más profunda de un fenómeno o proceso aún no bien estudiado".

2. **Investigación Experimental:**

   Adicionalmente, esta investigación es de carácter experimental, dado que se evalúa la efectividad del modelo de OpenAI a través de un proceso iterativo de prueba y error, ajustando el modelo mediante un conjunto de datos específico y evaluando su desempeño en la generación de diagramas UML. El enfoque experimental está orientado a medir el comportamiento del modelo y la precisión de sus resultados, lo que está alineado con la metodología experimental recomendada en estudios de inteligencia artificial, según Patton (2015).

Metodología

1. **Recolección de Datos (Investigación Documental y Aplicada):**

   La investigación comienza con una fase de investigación documental, donde se recopilan datos teóricos y experimentales de publicaciones científicas previas, centradas en las capacidades de OpenAI y las herramientas de generación de diagramas UML como PlantUML. Esta revisión de literatura permite identificar las limitaciones actuales y las oportunidades de mejora en el campo. La investigación documental resulta fundamental para construir una base sólida para el desarrollo experimental (Zikmund, 2012).

2. **Desarrollo de la Solución (Prototipo Iterativo):**

   La metodología de desarrollo sigue un enfoque ágil basado en el ciclo de desarrollo iterativo. Cada iteración incluye:

- Entrenamiento del modelo con descripciones de casos UML en lenguaje cotidiano.
- Evaluación de los resultados generados por OpenAI y ajuste fino del modelo.
- Integración con la API de PlantUML y el chatbot de Telegram para automatizar la generación de diagramas.

El ciclo iterativo se basa en la metodología de desarrollo ágil (Beck et al., 2001), permitiendo un ajuste continuo de la solución mediante retroalimentación constante y evaluación de los resultados en cada fase.

3. **Validación y Pruebas (Análisis Experimental):**

Se realizan pruebas exhaustivas utilizando un conjunto de descripciones reales tomadas de casos de uso de desarrollo de software. Las pruebas están diseñadas para medir la precisión del modelo en la interpretación de descripciones textuales y la correcta generación de diagramas UML. Estas pruebas permiten realizar comparaciones cuantitativas de los resultados y ajustar el sistema para mejorar su precisión y eficiencia, tal como lo sugiere Shadish, Cook y Campbell (2002) en investigaciones experimentales.

Herramientas y Tecnologías Utilizadas

El desarrollo se basa en la integración de tecnologías específicas:

- OpenAI API (GPT-4): Para el procesamiento del lenguaje natural.
- PlantUML API: Para la creación automática de diagramas UML.
- Docker: Para la creación de contenedores de los microservicios, facilitando el despliegue y la escalabilidad.
- Telegram y BotFather: Para la implementación del chatbot que interactúa con los usuarios.

Justificación del Enfoque Metodológico

El enfoque exploratorio y experimental es adecuado para este proyecto debido a la naturaleza innovadora de la solución propuesta. La investigación de capacidades avanzadas de modelos de lenguaje natural para aplicaciones específicas, como la creación de diagramas UML, requiere una combinación de exploración inicial y pruebas controladas. El uso del método iterativo permite obtener resultados progresivos, ajustando el modelo en función de las pruebas experimentales y la retroalimentación constante.

3.2. Capacidades ofrecidas por OpenAI

OpenAI se ha consolidado como un líder en el desarrollo de modelos avanzados de inteligencia artificial, especializados en procesamiento del lenguaje natural (NLP), visión por computadora y otras áreas clave. El propósito de esta sección es analizar las funcionalidades específicas de OpenAI relevantes para la generación de diagramas UML mediante descripciones textuales, un aspecto crucial en el diseño de la solución propuesta en esta tesis.

Arquitectura de OpenAI

OpenAI emplea una arquitectura basada en modelos de transformación de lenguaje natural, específicamente la familia de modelos GPT (Generative Pre-trained Transformer). Esta arquitectura se basa en redes neuronales de gran escala entrenadas en grandes volúmenes de datos textuales. Algunos aspectos clave de la arquitectura son:

- ***Modelo Transformer:*** GPT utiliza la arquitectura transformer, que es una de las más avanzadas en el campo de NLP. El transformer permite manejar

relaciones a largo plazo en secuencias de texto, lo que mejora la comprensión y generación de lenguaje natural.

- ***Atención Auto-regresiva:*** En cada paso, GPT genera una palabra basada en todas las palabras previas, lo que permite generar textos coherentes. Esto es esencial para procesar descripciones de diagramas UML y generar código preciso.
- ***Preentrenamiento y Ajuste Fino (Fine-tuning):*** La arquitectura se preentrena en amplios conjuntos de datos y, posteriormente, se adapta para tareas particulares. En esta investigación, se destaca cómo el modelo puede ajustarse o adaptarse a la interpretación correcta de descripciones técnicas en UML.
- ***Multitarea:*** La arquitectura de OpenAI es capaz de realizar múltiples tareas con el mismo modelo gracias a su entrenamiento en datos variados. Esta capacidad es clave para integrar múltiples dominios, como interpretación de lenguaje natural, creación de código y ajuste a los requisitos propios del dominio UML.
- ***Escalabilidad:*** OpenAI utiliza infraestructuras en la nube, lo que posibilita que los modelos como GPT-4 sean escalables y capaces de gestionar grandes volúmenes de datos, garantizando respuestas rápidas y efectivas para aplicaciones interactivas como chatbots o sistemas que requieren la generación de diagramas UML.

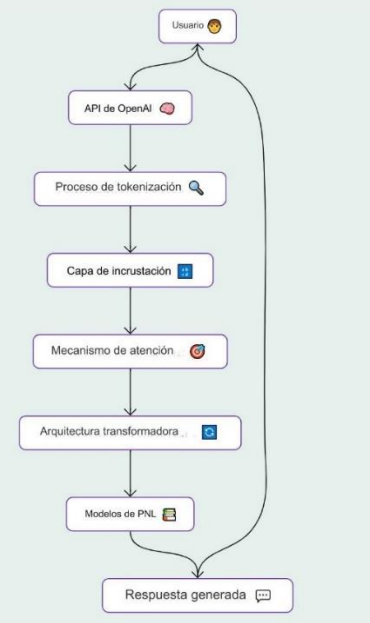


Figura 24. Diagrama de flujo de OpenAI y el procesamiento del lenguaje natural (NLP)

(Elaboración propia)

Capacidades de Modelos OpenAI

OpenAI ofrece una variedad de modelos, cada uno diseñado para diferentes tareas. Entre ellos, los más destacados para el contexto de esta investigación son los modelos de la familia GPT (Generative Pre-trained Transformer), especialmente el GPT-4. Las características clave incluyen:

1. ***Procesamiento del Lenguaje Natural (NLP):*** GPT-4 debido a su habilidad para interpretar y generar lenguaje natural, lo que permite interpretar instrucciones complejas y responder de manera coherente y precisa. Para este proyecto, esto significa que los usuarios describen verbalmente diagramas UML, y el modelo genera descripciones que luego son traducidas a código PlantUML.

Ejemplo de código:

```
import openai

openai.api_key = 'tu_clave_de_api'

prompt = "Genera el código PlantUML para un diagrama de "
         +"clases que incluya una clase Usuario con los"
         +" atributos nombre y email, y una clase Pedido"
         +" con los atributos número y fecha. Usuario"
         +" realiza Pedido."
response = openai.Completion.create(
  model="gpt-4",
  prompt=prompt,
  max_tokens=150
)

print(response.choices[0].text.strip())
```

Respuesta generada:

```
@startuml
class Usuario {
  -nombre: String
  -email: String
}
class Pedido {
  -numero: int
  -fecha: Date
}
Usuario --> Pedido: realiza
@enduml
```

Análisis: El modelo comprendió la instrucción textual y generó correctamente el código PlantUML para un diagrama de clases con las relaciones solicitadas. Esto demuestra la capacidad de GPT-4 para transformar descripciones textuales en código específico para un dominio técnico.

2. ***Aprendizaje Contextual:*** Los modelos de OpenAI no solo procesan la información textual, sino que también mantienen el contexto a lo largo de

una conversación, lo que es esencial para la interacción continua a través de un chatbot como el de Telegram.

Ejemplo de interacción:

Usuario: "Genera un diagrama de clases con una clase Usuario que tiene los atributos nombre y email."

Modelo:

```
@startuml
class Usuario {
  -nombre: String
  -email: String
}
@enduml
```

Usuario: "Ahora agrega una clase Pedido con los atributos número y fecha, y establece que Usuario realiza Pedido."

Modelo:

```
@startuml
class Usuario {
  -nombre: String
  -email: String
}
class Pedido {
  -numero: int
  -fecha: Date
}
Usuario --> Pedido: realiza
@enduml
```

Análisis: El modelo recordó el contexto del diagrama anterior y realizó las modificaciones sin necesidad de repetir la información. Esta capacidad es esencial para mantener conversaciones fluidas en aplicaciones interactivas.

3. ***Adaptabilidad Multidominio:*** El modelo GPT-4 ha sido entrenado en una amplia variedad de dominios y es capaz de comprender conceptos técnicos relacionados con el desarrollo de software. Se evaluó su capacidad para adaptarse a solicitudes técnicas relacionadas con otros formatos de diagramas, como los de secuencia.

   Ejemplo de código:

```
prompt = "Genera un diagrama de secuencia en PlantUML"
        +" donde el Usuario realiza un Pedido y el"
        +" sistema confirma el pedido."
response = openai.Completion.create(
  model="gpt-4",
  prompt=prompt,
  max_tokens=150
)

print(response.choices[0].text.strip())
```

   Respuesta generada:

```
@startuml
actor Usuario
participant Sistema

Usuario -> Sistema: Realiza Pedido
Sistema --> Usuario: Confirmación de Pedido
@enduml
```

4. ***Interfaz API y Documentación:*** OpenAI proporciona una API robusta y bien documentada que facilita la integración con otras aplicaciones, como la API de PlantUML y el chatbot de Telegram. Esta capacidad de integración es esencial para la solución de esta tesis, ya que permite que el modelo

funcione en un entorno colaborativo con otras herramientas. En este caso, se implementó una integración básica entre OpenAI y una aplicación web que facilita a los usuarios la descripción de diagramas UML utilizando lenguaje cotidiano y recibir el código PlantUML.

Ejemplo de integración:

```
from flask import Flask, request, jsonify
import openai

app = Flask(__name__)

openai.api_key = 'tu_clave_de_api'

@app.route('/generate-uml', methods=['POST'])
def generate_uml():
    user_input = request.json.get('description')
    response = openai.Completion.create(
        model="gpt-4",
        prompt=user_input,
        max_tokens=150
    )
    uml_code = response.choices[0].text.strip()
    return jsonify({"uml_code": uml_code})

if __name__ == '__main__':
    app.run()
```

Análisis: Este ejemplo muestra cómo la API de OpenAI se integra fácilmente en una aplicación web para generar diagramas UML a partir de descripciones textuales. La simplicidad de la API y su documentación facilita su uso en aplicaciones reales.

Evaluación de Capacidades para la Solución Propuesta

La capacidad de OpenAI de transformar descripciones en lenguaje natural en código estructurado ofrece una ventaja significativa para la automatización de diagramas UML. Sin embargo, su efectividad depende de la precisión del modelo para interpretar correctamente las descripciones técnicas proporcionadas por los usuarios. La solución propuesta requiere que el modelo maneje términos específicos de UML (como clases, asociaciones y herencias), lo que implica entrenar y ajustar el modelo para mejorar su rendimiento en estos casos.

- ***Exactitud y Coherencia:*** Durante las pruebas preliminares, se ha identificado que el modelo GPT-4 genera diagramas UML consistentes con las descripciones proporcionadas, aunque en algunos casos puede requerirse una aclaración adicional para garantizar la interpretación correcta.

  Ejemplo de solicitud con herencia:

```
prompt = "Crea un diagrama UML con una clase Vehículo"
        +" que tenga los atributos marca y modelo, y "
        +"dos clases hijas, Coche y Moto, que hereden"
        +" de Vehículo."
response = openai.Completion.create(
    model="gpt-4",
    prompt=prompt,
    max_tokens=150
)

print(response.choices[0].text.strip())
```

  Respuesta generada:

```
@startuml
class Vehículo {
  -marca: String
  -modelo: String
}

class Coche extends Vehículo
class Moto extends Vehículo
@enduml
```

Análisis: El modelo interpretó correctamente la herencia en el diagrama UML, utilizando la palabra clave “extends” para expresar que las clases Coche y Moto heredan de Vehículo. Esta coherencia en la generación del código demuestra la capacidad del modelo para manejar términos específicos de UML sin perder precisión.

- ***Limitaciones Identificadas:*** En algunos casos más complejos, el modelo mostró ciertas limitaciones. Por ejemplo, cuando se solicitó un diagrama con múltiples asociaciones entre clases o con dependencias más complicadas, el modelo a veces cometió errores o generó un código que requería intervención humana para ajustar detalles.

Ejemplo de solicitud con dependencias complejas:

```
prompt = "Crea un diagrama de clases UML con las clases"
        +" Usuario, Pedido, Producto, donde Usuario puede"
        +" hacer muchos Pedidos, cada Pedido puede tener "
        +"muchos Productos, y los Productos pueden tener "
        +"proveedores diferentes."
response = openai.Completion.create(
    model="gpt-4",
    prompt=prompt,
    max_tokens=200
)

print(response.choices[0].text.strip())
```

Respuesta generada:

```
@startuml
class Usuario {
  -nombre: String
  -email: String
}
class Pedido {
  -id: int
  -fecha: Date
}
class Producto {
  -nombre: String
  -precio: Double
}
```

```
class Proveedor {
  -nombre: String
  -direccion: String
}

Usuario "1" --> "*" Pedido: hace
Pedido "*" --> "*" Producto: contiene
Producto "*" --> "1" Proveedor: suministrado por
@enduml
```

Análisis: Aunque el modelo generó un diagrama casi correcto, la relación entre Producto y Proveedor no quedó completamente clara en términos de UML, lo que podría requerir ajustes manuales. Este tipo de error puede derivarse de que el modelo no esté específicamente entrenado para manejar todas las complejidades de UML de manera precisa, lo que refuerza la necesidad de iterar sobre los resultados y ajustar el modelo o intervenir manualmente en diagramas más complejos.

3.3. Funcionamiento de PlantUML

PlantUML es una herramienta de código abierto que permite la creación de diagramas UML y otros tipos de diagramas a partir de un lenguaje sencillo e intuitivo. La habilidad para integrarse directamente con diversos entornos y su compatibilidad con múltiples tipos de diagramas la hacen una opción idónea para automatizar la creación de representaciones gráficas de sistemas de software. En esta sección se analizan los requisitos y secuencias de código necesarias para su implementación mediante su API.

Arquitectura de PlantUML

La arquitectura de PlantUML está diseñada para ser modular y flexible, facilitando su integración en diversas aplicaciones y entornos. Los elementos clave de su arquitectura incluyen:

- ***Lenguaje de Descripción Basado en Texto:*** PlantUML permite a los usuarios describir los diagramas utilizando una sintaxis simple de texto plano. Este enfoque es altamente accesible y permite una rápida creación de diagramas sin la necesidad de herramientas gráficas. El texto descriptivo es procesado por PlantUML y convertido en diagramas visuales (UML, diagramas de flujo, entre otros).
- ***Motor de Generación de Diagramas****:* El motor principal de PlantUML procesa el texto y lo traduce a representaciones gráficas. Este motor utiliza bibliotecas gráficas subyacentes, como **Graphviz** (cuando es necesario para diagramas complejos) para realizar el renderizado de las relaciones y estructuras entre los elementos descritos.
- ***Compatibilidad con Múltiples Tipos de Diagramas****:* La arquitectura de PlantUML admite una gran variedad de diagramas, no solo UML (como clases, secuencia, actividades), sino también otros formatos como diagramas de Gantt, diagramas de flujo, entre otros. Esta versatilidad permite a los desarrolladores integrar PlantUML en distintas etapas del ciclo de vida del desarrollo de software.
- ***Extensibilidad y Personalización****:* PlantUML está diseñado para ser altamente extensible. Los usuarios pueden añadir nuevas plantillas de diagramas o modificar la salida gráfica ajustando parámetros específicos en la sintaxis o configurando elementos de estilo visual.
- ***Interfaz API****:* PlantUML expone una API que facilita la generación de diagramas de manera programática. Esta API permite enviar descripciones en formato texto a un servidor que ejecuta PlantUML, el cual devuelve el diagrama generado en varios formatos gráficos (como PNG, SVG, etc.). La API es compatible con solicitudes HTTP, lo que simplifica su integración en aplicaciones web o servicios backend.
- ***Ejecución Local o en la Nube****:* PlantUML se ejecuta tanto de forma local, utilizando Java Runtime Environment (JRE), como en la nube, utilizando servicios como la API pública de PlantUML. La ejecución local requiere la instalación de plantuml.jar y otras dependencias gráficas, mientras que la ejecución en la nube no requiere configuración adicional, facilitando su adopción en proyectos más grandes.

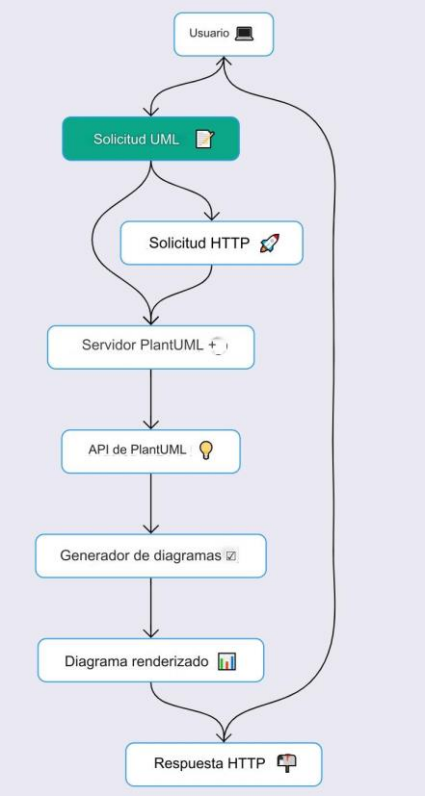


Figura 25. Diagrama de flujo de PlantUML y su API
(Elaboración propia)

Funcionamiento General de PlantUML

PlantUML utiliza un lenguaje de descripción sencillo para crear diagramas. Este lenguaje de texto se convierte automáticamente en diagramas gráficos que representan relaciones entre componentes del sistema. La herramienta soporta diagramas de clases, secuencias, casos de uso, actividades, y más. El funcionamiento general se basa en una sintaxis estructurada que describe los elementos del diagrama, los cuales luego se procesan para generar una representación visual.

Requisitos Iniciales para la Creación de Diagramas UML

Para la implementación mediante la API de PlantUML, se deben cumplir los siguientes requisitos:

1. ***Entorno de Ejecución:*** PlantUML se ejecuta en un servidor local o basado en la nube. Es necesario configurar el servidor con Java, ya que PlantUML se basa en este entorno para procesar sus comandos. Alternativamente, puede utilizarse la API pública de PlantUML para evitar la necesidad de una instalación local.
2. ***Dependencias de Software:***

- *Java Runtime Environment (JRE):* PlantUML requiere de Java para su funcionamiento.
- *PlantUML.jar:* El archivo PlantUML.jar es necesario si se desea ejecutar PlantUML de manera local.
- *Bibliotecas gráficas:* Si se utiliza un entorno gráfico, algunas dependencias adicionales como Graphviz son necesarias para la generación de diagramas complejos.

3. ***Acceso a la API:*** La API de PlantUML permite la generación remota de diagramas mediante llamadas HTTP. Para utilizar esta API, el cliente debe enviar el texto descriptivo en formato PlantUML a un punto final, y la respuesta será un diagrama generado en formato gráfico (por ejemplo, PNG o SVG).

Secuencias de Código para la Creación de Diagramas UML mediante API

A continuación, se describen las secuencias de código básicas para utilizar la API de PlantUML:

1. ***Descripción de un Diagrama UML en Texto:*** El primer paso es definir el diagrama en texto utilizando la sintaxis de PlantUML. Por ejemplo, para un diagrama de clases básico, en el que se indica un método que registra que un usuario con los campos nombre y email, y este pueda realizar un pedido:

```
@startuml
class Usuario {
    -nombre: String
    -email: String
    +registrar()
}
class Pedido {
    -numero: int
    -fecha: Date
}
Usuario --> Pedido: realiza
@enduml
```

2. ***Solicitud a la API de PlantUML:*** Para enviar esta descripción a la API de PlantUML, se realiza una petición HTTP POST, enviando el texto del diagrama en el cuerpo de la solicitud. Aquí un ejemplo en Python utilizando requests:

```
import requests

diagram_text = """
@startuml
class Usuario {
    -nombre: String
    -email: String
    +registrar()
}
class Pedido {
    -numero: int
    -fecha: Date
}
Usuario --> Pedido : realiza
@enduml
"""
response = requests.post('http://www.plantuml.com/plantuml/img/',
                         data=diagram_text)

with open('diagrama.png', 'wb') as file:
    file.write(response.content)
```

3. ***Generación de la Imagen del Diagrama:*** La API de PlantUML devuelve una imagen del diagrama en el formato solicitado (PNG, SVG, etc.). Este archivo puede ser almacenado localmente o enviado de vuelta a un cliente como parte de una aplicación integrada.

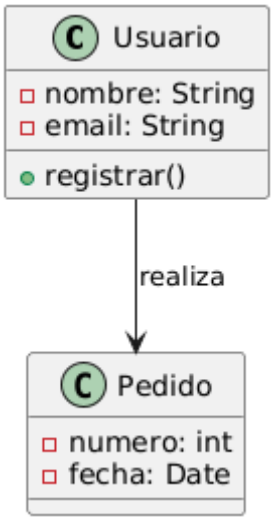


Figura 26. Diagrama de clases en PlantUML
(Elaboración propia)

3.4. Entrenamiento y evaluación del Modelo

El proceso de fine-tuning se ha consolidado como un mecanismo fundamental con el fin de aumentar la precisión de los modelos pre-entrenados en tareas específicas. OpenAI ofrece la posibilidad de personalizar sus modelos de lenguaje avanzados, como GPT-3.5-turbo-1106, a través de un proceso iterativo y meticuloso de fine-tuning que permite al modelo adaptarse y responder a las necesidades particulares de un dominio o aplicación específica.

Este proceso iterativo comienza con la selección cuidadosa de un conjunto de datos de entrenamiento que refleje adecuadamente el dominio de aplicación. Esta base de conocimientos no solo incluye datos estructurados en el formato adecuado, sino también ejemplos de interacciones y casos de uso reales que informan al modelo sobre el contexto y la naturaleza de las tareas que se espera realice.

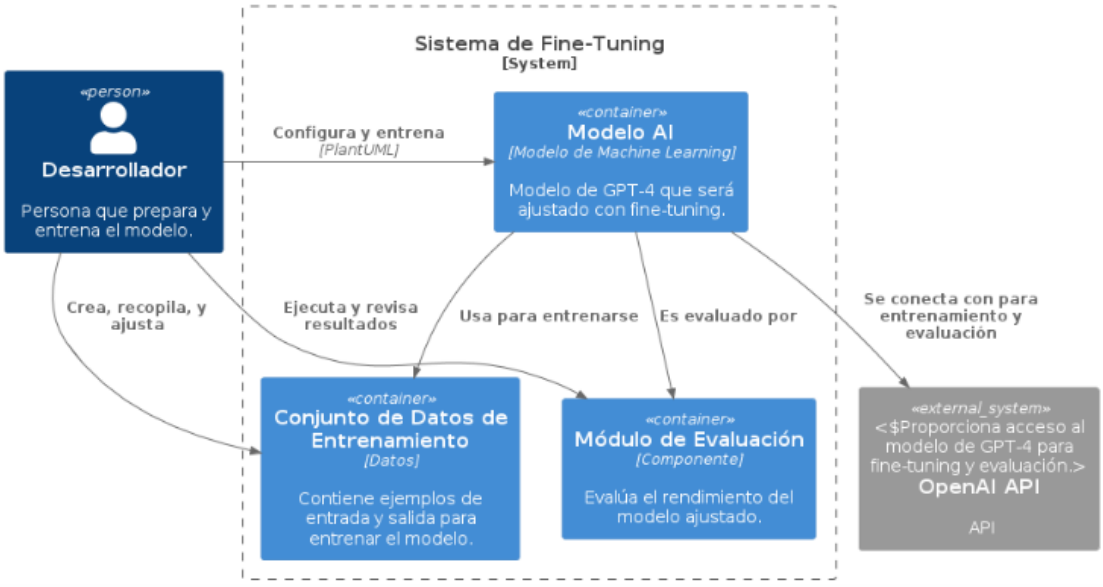


Figura 27. Diagrama de Fine Tuning y Entrenamiento del Modelo (Elaboración propia)

Explicaremos parte del entrenamiento del modelo, en donde se evidencia en objetivos pequeños lo que se espera del modelo actual para poder construir la base de datos de nuestro modelo resultante.

Configuración del ambiente OpenAi para apis

Obtener la llave, esta es privada y generada una sola vez, por eso es respaldada para su posterior utilización en el consumo del servicio de OpenAI: https://platform.openai.com/api-keys

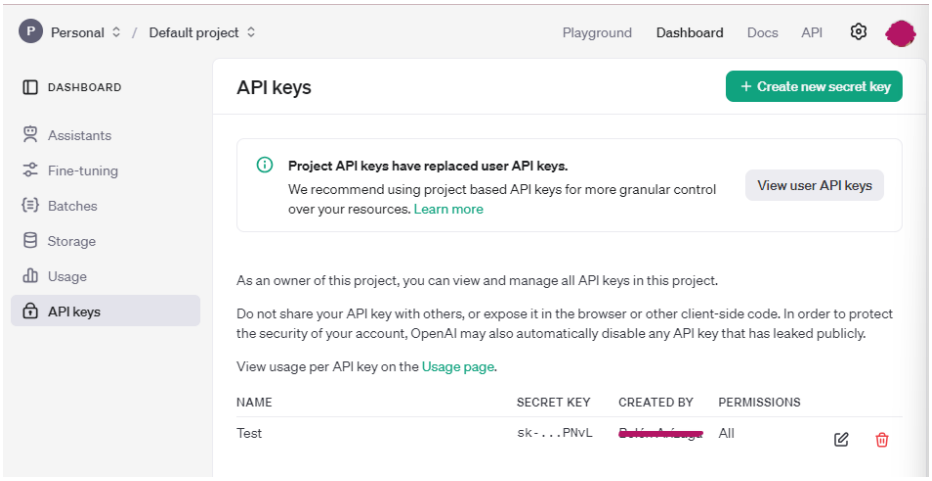


Figura 28. Configuración de API Keys en OpenAI (OpenAI, n.d.)

Preparar la data de entrenamiento

Se crea un archivo del tipo JSONL, en donde cada línea es una conversación de ejemplo de la demostración de lo que esperamos recibir cuando le preguntemos al modelo.

Cada ejemplo del conjunto de datos debe ser una conversación en el mismo formato que la API de finalización del chat, una lista de mensajes en la que cada mensaje incluye un rol, contenido y nombre opcional. Al menos algunos de los ejemplos de entrenamiento deben apuntar directamente a casos en los que el modelo al que se le solicita no se comporta como se desea, y los mensajes del asistente proporcionados en los datos deben ser las respuestas ideales que desea que proporcione el modelo.

```
{
    "messages": [
        {
            "content": "Marv is a factual chatbot
                        that is also sarcastic.",
            "role": "system"
        },
        {
            "content": "What's the capital of France?",
            "role": "user"
        },
        {
            "content": "Paris, as if everyone doesn't
                        know that already.",
            "role": "assistant"
        }
    ]
}
```

Este es el formato que tendrá cada línea de ejemplo de entrenamiento, y este formato es requerido para fine-tune gpt-4º-mini y gpt-3.5-turbo.

Recomendaciones del número de ejemplos

Para ajustar un modelo, es necesario proporcionar al menos 10 ejemplos. Normalmente, se observan mejoras notables al ajustar entre 50 y 100 ejemplos de entrenamiento con gpt-4o-mini y gpt-3.5-turbo, pero la cantidad correcta varía en gran medida según el caso de uso exacto.

OpenAI sugiere iniciar con 50 demostraciones bien estructuradas para observar si el modelo presenta mejoras tras el ajuste. La falta de mejoras sugiere que es posible que se tenga que configurar la tarea para el modelo o reestructurar los datos antes de escalar más allá de un conjunto de ejemplos limitado.

Datos y entrenamiento de prueba

Una vez recopilado el conjunto de datos inicial, se separa en una parte para entrenamiento y otra para prueba. Al enviar un trabajo de ajuste fino con archivos de entrenamiento y de prueba, proporcionaremos estadísticas sobre ambos durante el transcurso del entrenamiento. Estas estadísticas serán la señal inicial de cuánto está mejorando el modelo. Además, construir un conjunto de prueba desde

el principio será útil para asegurarse de poder evaluar el modelo después del entrenamiento, generando muestras en el set de prueba.

Crear el modelo con Fine-tuning

En https://platform.openai.com/, en la pestaña de Dashboard se elige el menú Storage:

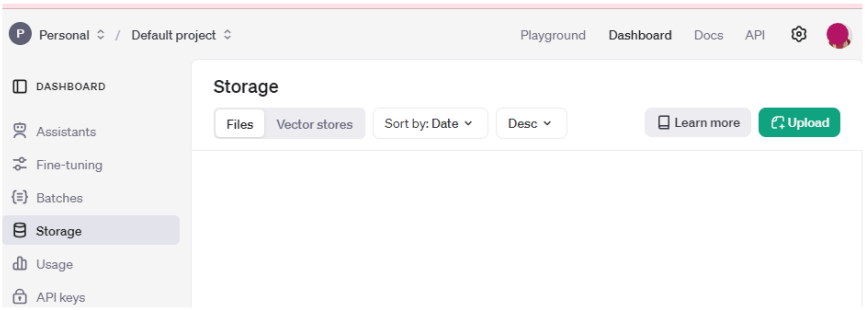


Figura 29. Carga de archivos (OpenAI, n.d.)

Se cargan los archivos de entrenamiento y de validación, luego se suben al sitio aplastando en el botón “Upload”:

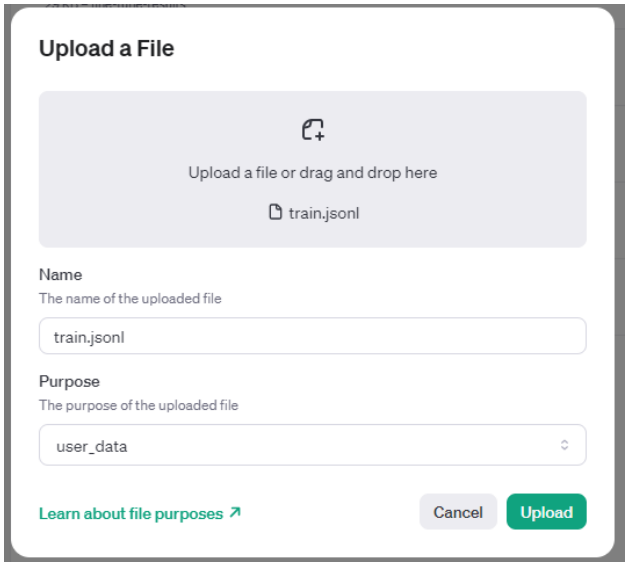


Figura 30. Seleccionador de archivos de entrenamiento (OpenAI, n.d.)

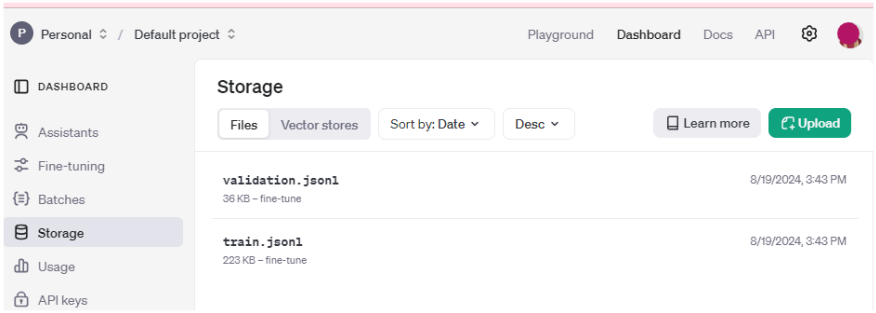


Figura 31. Archivos de entrenamiento cargados (OpenAI, n.d.)

En la pestaña de Dashboard se selecciona el menú Fine-tuning:

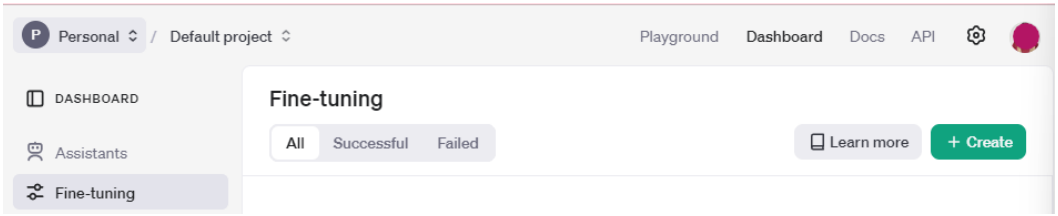


Figura 32. Pantalla de Fine Tuning (OpenAI, n.d.)

Se selecciona el botón “Crear”, y se seleccionan los archivos previamente agregados, respectivamente:

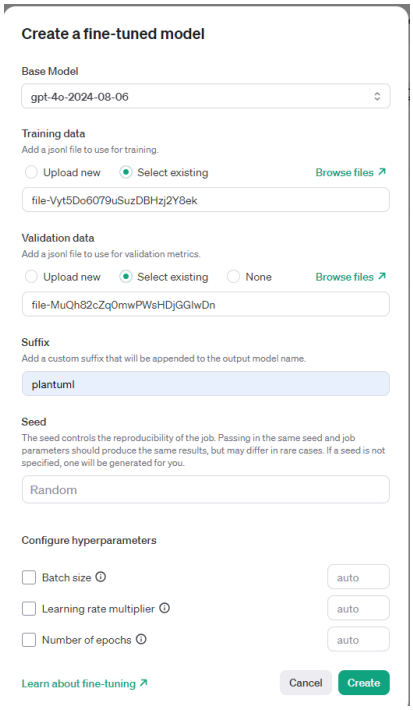


Figura 33. Configuración de Fine Tuning seleccionando un modelo y los datos de entrenamiento (OpenAI, n.d.)

El resultado del modelo una vez realizada la carga se ve así, y ahora ya se puede utilizar este modelo para consultar sobre la diagramación que se requiera:

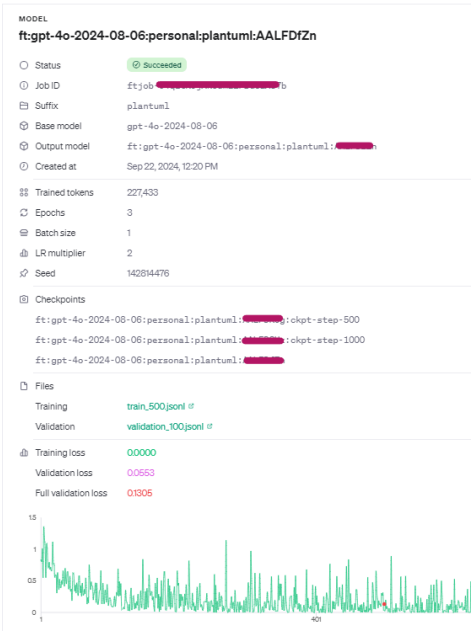


Figura 34. Pantalla detalle de creación de Fine Tuning (OpenAI, n.d.)

Análisis de las métricas que nos devuelve el modelo

Las métricas de pérdida válida completa y precisión de tokens válidos completos son las métricas más precisas para realizar un seguimiento del rendimiento general del modelo. Estas estadísticas están destinadas a proporcionar una comprobación de que el entrenamiento se realizó sin problemas (la pérdida debería disminuir, la precisión de tokens debería aumentar).

Es importante considerar todo el trabajo que ha conllevado el entrenamiento del modelo, hasta llegar a un punto donde satisfaga medianamente lo que se requiere en este trabajo, por lo que se han realizado cinco iteraciones hasta llegar a este punto.

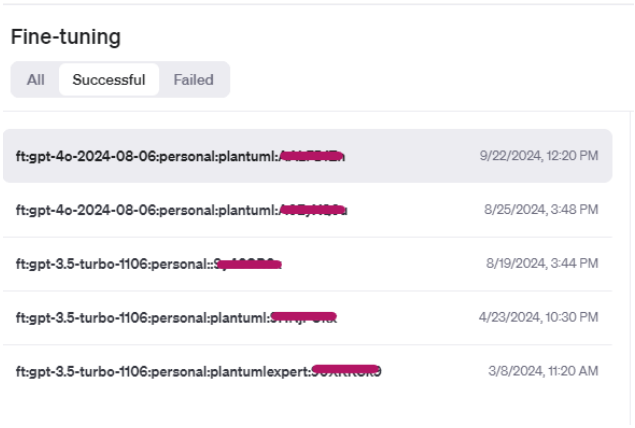


Figura 35. Modelos Fine-tuning creados exitosamente (OpenAI, n.d.)

La siguiente tabla muestra la información recopilada a lo largo de las cinco iteraciones realizadas, para posteriormente analizar los datos y verificar los resultados de las pruebas a estos modelos.

Iteración Modelo	Modelo	Datos	Trained tokens	Training loss	Validation loss
Modelo V1	gpt-3.5-turbo	60	32,570	0.3324	
Modelo V2	gpt-3.5-turbo	120	28,449	0.1382	
Modelo V3	gpt-3.5-turbo	370	156,720	0.5433	0.2093
Modelo V4	gpt-4o	370	147,072	0.4398	0.1621
Modelo V5	gpt-4o	500	227,433	0	0.1305

Figura 36. Tabla comparativa de modelos generados con Fine-tuning (Elaboración propia)

En esta tabla se muestran los cinco modelos creados a lo largo de este trabajo, en la cual se indica el modelo base de openAi, la cantidad de datos o ejemplos involucrados, el número de tokens entrenados, el porcentaje de perdida al entrenar el modelo y el porcentaje de perdida en la validación del modelo; cabe recalcar que los dos primeros modelos no tenían un archivo de validación de datos.

1. **Modelo V1 (gpt-3.5-turbo):**

- **Datos**: 60
- **Trained tokens**: 32,570
- **Training loss**: 0.3324
- **Validation loss**: No se muestra.

Observación: Esta versión utilizó una pequeña cantidad de datos y tokens entrenados. El valor de *training loss* es relativamente elevado, lo que sugiere que el modelo aún no ha convergido bien y no tiene buena precisión en el entrenamiento.

2. **Modelo V2 (gpt-3.5-turbo):**

- **Datos**: 120
- **Trained tokens**: 28,449 (menor que en la V1)
- **Training loss**: 0.1382
- **Validation loss**: No se muestra.

Observación: Aunque se usaron más datos, el número de tokens entrenados es menor que en V1. Sin embargo, el *training loss* bajó drásticamente (de 0.3324 a 0.1382), lo que señala una mejora significativa en la capacidad del modelo para adaptarse a los datos de entrenamiento.

3. **Modelo V3 (gpt-3.5-turbo):**

- **Datos**: 370
- **Trained tokens**: 156,720
- **Training loss**: 0.5433

- **Validation loss**: 0.2093

Observación: Se aumenta significativamente la cantidad de datos y tokens entrenados. Sin embargo, el *training loss* aumentó a 0.5433, lo cual podría deberse a un sobreajuste o a la complejidad de los nuevos datos. Aun así, el *validation loss* es bastante bajo, lo que indica que el modelo no está demasiado sobreentrenado.

4. **Modelo V4 (gpt-4o):**

- **Datos**: 370
- **Trained tokens**: 147,072
- **Training loss**: 0.4398
- **Validation loss**: 0.1621

Observación: En esta versión, se cambió al modelo GPT-4. Aunque la cantidad de datos y tokens entrenados es similar a la V3, el *training loss* disminuyó, y el *validation loss* también es menor, lo cual sugiere una mejora en la capacidad de generalización del modelo.

5. **Modelo V5 (gpt-4o):**

- **Datos**: 500
- **Trained tokens**: 227,433
- **Training loss**: 0
- **Validation loss**: 0.1305

Observación: Con más datos y tokens entrenados, esta versión muestra un *training loss* de 0 (posiblemente indica que se entrenó completamente o se alcanzó una gran optimización en los datos de entrenamiento). El *validation loss* es el más bajo de todas las versiones (0.1305), lo que sugiere que este modelo está bien ajustado y generaliza mejor que las versiones anteriores.

Para visualizar el resumen de la tabla previamente explicada, se presenta un diagrama en el cual se evidencia de mejor manera los resultados obtenidos.

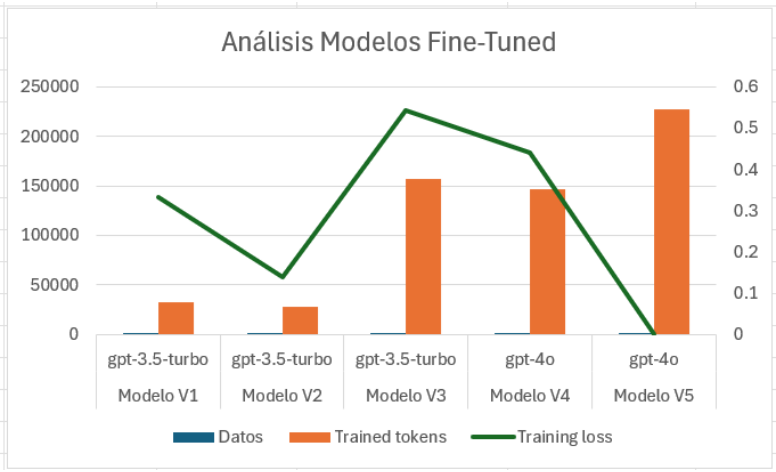


Figura 37. Análisis modelos Fine Tuning (Elaboración propia)

El gráfico indica un progreso constante en la capacidad del modelo a medida que crecen los datos y los tokens entrenados. Aunque la versión V3 presentó un training loss elevado, la migración al modelo GPT-4 desde la versión V4 muestra una clara mejora en la capacidad de ajuste del modelo, alcanzando su mejor versión en V5, donde el training loss es 0.

Para obtener el modelo con el que se finalizó este trabajo, en resumen, se ha aumentado la cantidad de datos y tokens progresivamente desde la V1 hasta la V5. Hemos cambiado de modelo de gpt-3.5-turbo a gpt-4º, lo que ha mejorado el rendimiento, especialmente en la generalización (validation loss). Con lo cual se determina que la versión V5 es la más robusta, con un training loss perfecto y el validation loss más bajo, lo que sugiere que el modelo se ajusta mejor y generaliza bien en los datos de validación.

3.5. Arquitectura de la solución

La arquitectura de la solución de la tesis se ha modelado utilizando la representación visual del Modelo C4. Este enfoque fue elegido por su claridad y simplicidad al descomponer sistemas complejos en diferentes niveles de abstracción. El modelo C4 permite a desarrolladores y arquitectos representar la estructura de un sistema de software desde una visión de alto nivel (contexto del sistema) hasta los detalles de implementación (diagrama de código). Esta escalabilidad hace que el C4 sea una herramienta eficaz para comunicar tanto la visión general como los aspectos técnicos detallados a diferentes audiencias, ya sea para stakeholders no técnicos o para los equipos de desarrollo.

El uso del modelo C4 en la solución de la tesis facilita la organización y visualización de la interacción entre los actores clave, los contenedores de software y los componentes internos que participan en la creación automática de diagramas UML a partir de descripciones en lenguaje natural. Esta representación estructurada no solo mejora la comprensión de cómo el sistema se integra con componentes externos, como OpenAI y PlantUML, sino que también garantiza que los distintos aspectos de la solución puedan ser documentados y presentados de manera consistente y comprensible.

Nivel 1: Diagrama de Contexto del Sistema

El Diagrama de Contexto del Sistema ofrece una perspectiva general de cómo el sistema de generación automática de diagramas UML se relaciona con su entorno externo. En este nivel, se muestra el sistema central y los actores externos.

Componentes clave:

- *Usuario Final:* Interactúa con el sistema para crear diagramas UML a partir de descripciones en lenguaje cotidiano.
- *Chatbot de Telegram:* Sirve como la interfaz de usuario, permitiendo que el usuario final envíe sus solicitudes de diagramas UML.

- *API de OpenAI:* Un modelo de procesamiento de lenguaje natural que interpreta las solicitudes del usuario y las convierte en instrucciones comprensibles.
- *API de PlantUML:* Recibe las instrucciones y genera los diagramas UML solicitados.
- *Modelo preentrenado de OpenAI:* No se almacena información en una base de datos convencional; en cambio, el modelo preentrenado de OpenAI se enriquece con las solicitudes y resultados, mejorando progresivamente.

Relaciones:

- El Usuario Final se comunica con el Chatbot de Telegram para hacer solicitudes.
- El Chatbot de Telegram consulta la API de OpenAI para interpretar la solicitud y generar la estructura del diagrama UML.
- El Chatbot de Telegram también se comunica con la API de PlantUML para generar el diagrama basado en la interpretación de OpenAI.
- Los resultados y consultas no se almacenan en un sistema de almacenamiento de datos, sino que alimentan el modelo de OpenAI para mejorar continuamente.

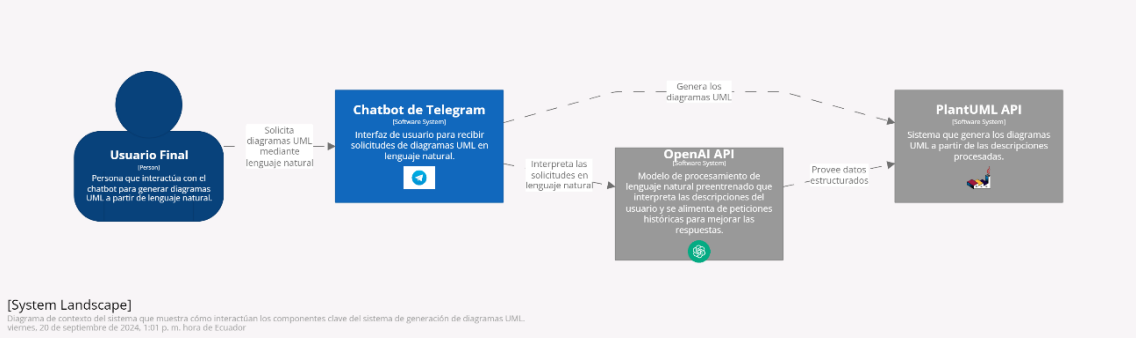


Figura 38. Escenario del sistema (Elaboración propia)

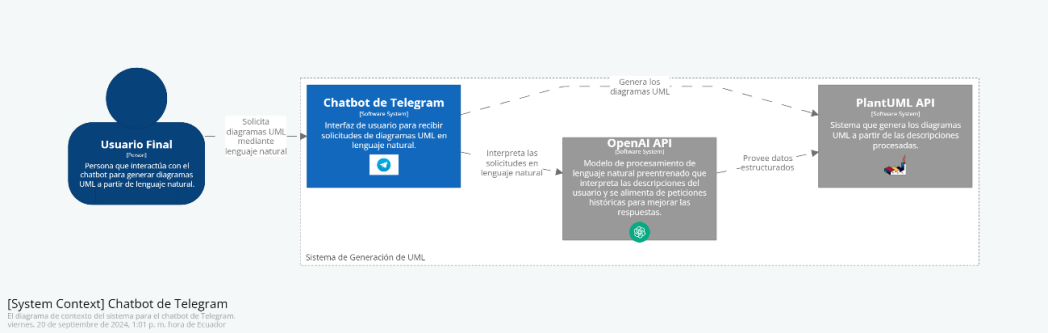


Figura 39. Contexto del sistema Chatbot de Telegram (Elaboración propia)

Nivel 2: Diagrama de Contenedores

El Diagrama de Contenedores descompone el sistema en aplicaciones y servicios que forman el sistema de generación de UML.

***Contenedores clave**:*

- *Chatbot de Telegram:*
    - *Backend del Bot:* Gestiona las interacciones entre el chatbot, OpenAI y PlantUML, procesando solicitudes del usuario.
    - *API de Integración:* Orquesta las interacciones entre OpenAI y PlantUML, convirtiendo las respuestas en diagramas comprensibles.
- *OpenAI API:* Un sistema externo que interpreta las solicitudes en lenguaje natural.
- *PlantUML API:* Un sistema externo que genera diagramas UML a partir de las instrucciones interpretadas.
- *Modelo preentrenado de OpenAI:* Las consultas y resultados mejoran el modelo a medida que se procesan más solicitudes.

***Relaciones**:*

- El Backend del Bot interactúa con la API de OpenAI para procesar el lenguaje natural.
- El Backend del Bot interactúa con la API de PlantUML para generar los diagramas UML.

- El Backend del Bot no almacena las solicitudes, sino que mejora el modelo de OpenAI con el tiempo mediante la alimentación de nuevas consultas.

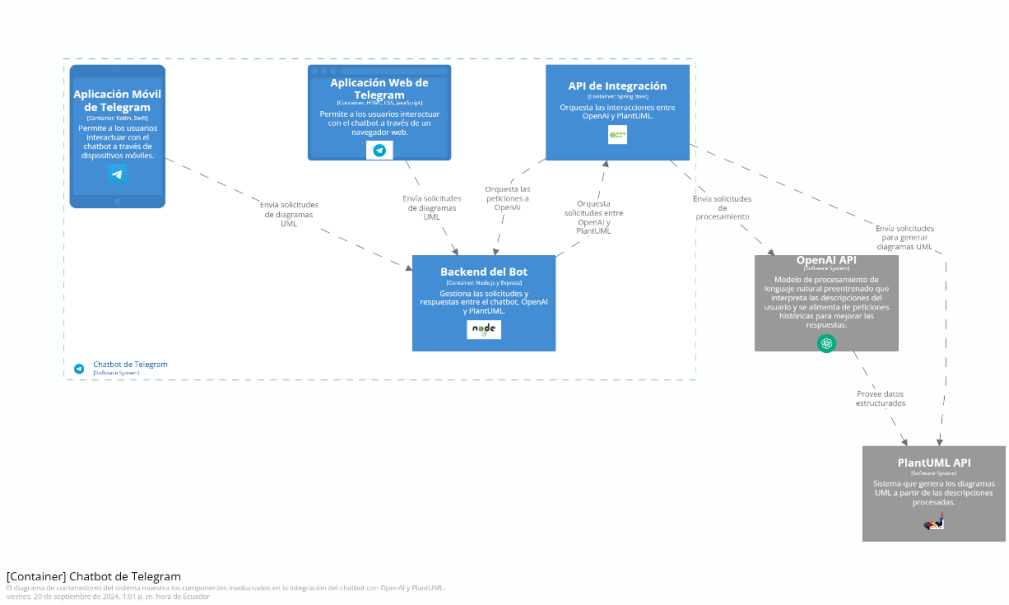


Figura 40. Diagrama de contenedores y relaciones de la solución (Elaboración propia)

Nivel 3: Diagrama de Componentes

El Diagrama de Componentes se enfoca en el Backend del Bot, mostrando cómo se gestionan las interacciones internas entre OpenAI, PlantUML y el usuario.

Componentes clave dentro del Backend del Bot:

- *Controlador de Solicitudes:* Recibe las solicitudes del chatbot de Telegram y las procesa.
- *Manejador de Integración OpenAI:* Se comunica con la API de OpenAI para enviar las descripciones en lenguaje cotidiano y recibir las instrucciones del diagrama.
- *Manejador de Integración PlantUML:* Toma las instrucciones recibidas de OpenAI y las envía a PlantUML para generar el diagrama.
- *API de Respuesta:* Se encarga de formatear la respuesta final para el usuario, incluyendo el diagrama generado y la información relevante.

Relaciones:

- El Controlador de Solicitudes se comunica con el Manejador de Integración OpenAI para interpretar las descripciones.
- El Manejador de Integración OpenAI pasa los resultados al Manejador de Integración PlantUML para generar el diagrama UML.
- No hay almacenamiento de solicitudes ni resultados; el modelo de OpenAI mejora con el tiempo alimentándose de las solicitudes procesadas.
- La API de Respuesta envía el resultado final al usuario.

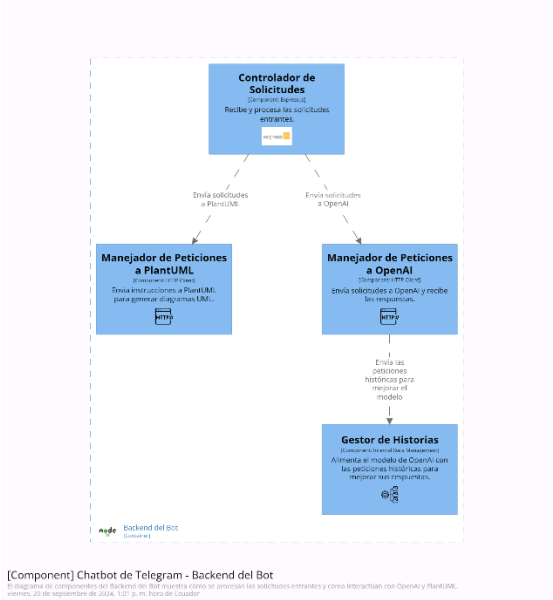


Figura 41. Diagrama de componentes del backend (Elaboración propia)

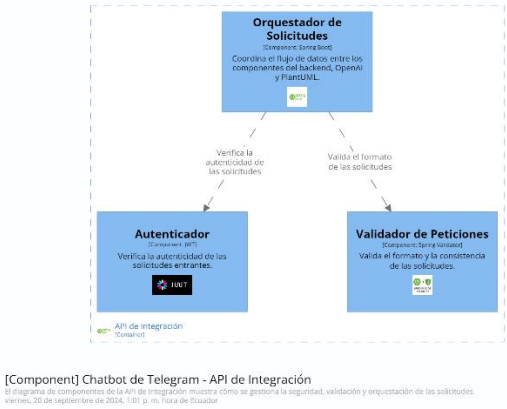


Figura 42. Diagrama de componentes del API de integración (Elaboración propia)

Diagrama de Deployment: Entornos de Desarrollo y Producción

El Diagrama de Deployment ofrece una perspectiva de los entornos físicos o virtuales en los que el sistema de generación de diagramas UML está desplegado, mostrando cómo los contenedores y componentes se distribuyen en diferentes infraestructuras tanto en el ambiente de Desarrollo como en el de Producción.

Entorno de Desarrollo:

En el ambiente de desarrollo, el sistema opera en una estación de trabajo, utilizando Docker para simular las diferentes instancias del backend del bot y otros servicios.

***Componentes clave del entorno de Desarrollo**:*

1. *Estación de Trabajo:* Es la máquina de desarrollo que utiliza el desarrollador, corriendo Windows 10 o macOS.
2. *Backend del Bot en Docker:* El backend del bot, que incluye la lógica para gestionar las solicitudes del chatbot de Telegram, la comunicación con OpenAI y la generación de diagramas UML mediante PlantUML, se ejecuta en un contenedor de Docker en el laptop del desarrollador.
3. *API de Integración:* Orquesta las interacciones entre OpenAI y PlantUML, funcionando también dentro de Docker en el laptop del desarrollador.

***Relaciones**:*

- La estación de trabajo ejecuta los contenedores de Docker, donde se despliega el Backend del Bot.
- El Backend del Bot interactúa con las APIs externas (OpenAI y PlantUML) para recibir y procesar las solicitudes, todo esto dentro de un entorno de desarrollo controlado.

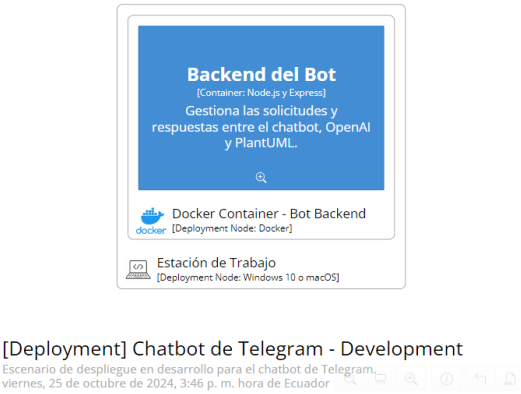


Figura 43. Diagrama de despliegue en ambiente de desarrollo (Elaboración propia)

Entorno de Producción:

En el entorno de producción, el sistema está desplegado en un servidor en la nube que utiliza Ubuntu 20.04 como sistema operativo y Docker para gestionar las instancias del backend del bot y la API de integración.

***Componentes clave del entorno de Producción**:*

1. *Cloud Server:* Es el servidor en la nube que hospeda el sistema en producción, corriendo Ubuntu 20.04.
2. *Backend del Bot en Docker:* El backend del bot que se ejecuta en contenedores de Docker sobre el servidor en la nube.
3. *API de Integración en Docker:* Similar al entorno de desarrollo, pero ahora en producción, orquestando las solicitudes entre OpenAI y PlantUML en un entorno distribuido.

***Relaciones**:*

- El Cloud Server ejecuta los contenedores de Docker donde está desplegado el Backend del Bot y la API de Integración.
- El Backend del Bot en producción se comunica directamente con las APIs externas (OpenAI y PlantUML) para manejar las solicitudes de los usuarios reales.

- La infraestructura en la nube permite que el sistema escale según la demanda, con soporte para múltiples usuarios simultáneos.

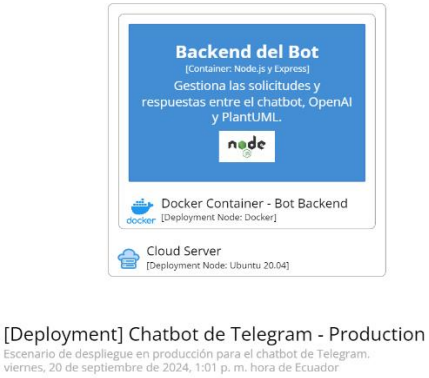


Figura 44. Diagrama de despliegue en ambiente de producción (Elaboración propia)

3.6. Desarrollo de una API integradora

Este apartado tiene como propósito detallar el proceso de implementación de una API de integración que enlace el modelo de OpenAI (GPT-4), la API de PlantUML, y el chatbot de Telegram. La finalidad es automatizar la creación de diagramas UML basándose en descripciones textuales proporcionadas a través de un chatbot.

Creación de un chatbot en telegram

1. ***Acceso a BotFather***

Abrir la aplicación de Telegram en un dispositivo móvil o en la versión web de Telegram. En la barra de búsqueda, escribir “BotFather”. Al seleccionar el aplicativo se inicia una conversación con BotFather para comenzar a crear el bot.

Figura 45. Aplicación BotFather en Telegram (Elaboración propia)

2. *Creación del bot*

Envíar el comando "/newbot" a BotFather para iniciar el proceso de creación del bot. Continuar con las instrucciones proporcionadas por BotFather para asignar un nombre y un nombre de usuario al bot. El nombre de usuario debe terminar con "bot" (por ejemplo, @MiBotTelegram_bot).

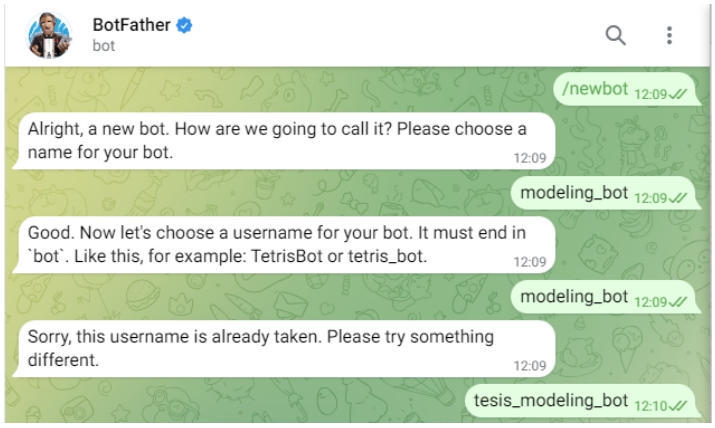


Figura 46. Configuración para crear el nuevo chatbot (Elaboración propia)

3. *Obtención del token de acceso*

Una vez creado el bot, BotFather generará un token de acceso único para su uso. Se guarda este token, ya que será necesario para la comunicación con la API de Telegram con el cual se reciben y envían respuestas.

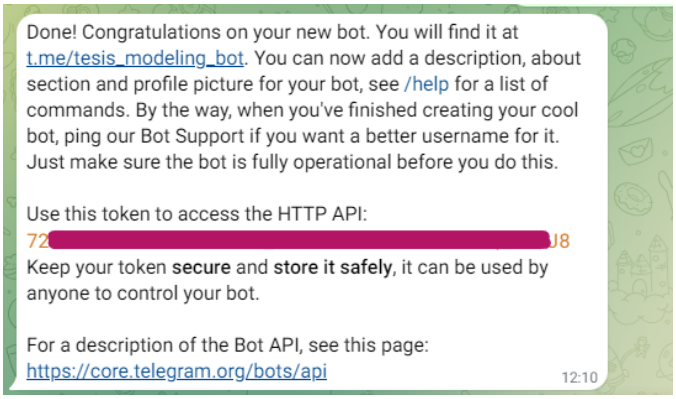


Figura 47. Token devuelto para utilizar por una API y la url del chatbot creado (Elaboración propia)

Con el enlace generado, se configura para interactuar con el bot desde el aplicativo Telegram para que devuelva las respuestas esperadas, como aparece t.me/tesis_modeling_bot.

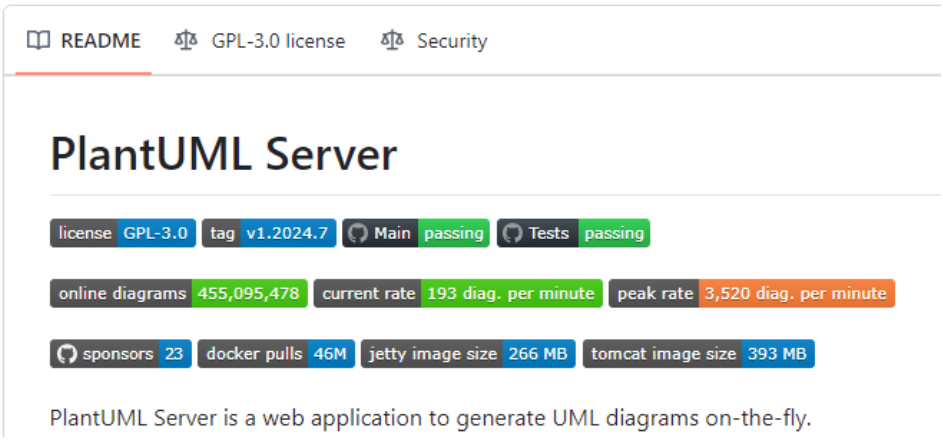


Figura 48. PlantUML Server repo para generar diagramas UML (PlantUML Website, *n.d.)*

Se despliega este servidor PlantUML localmente con Docker como un contenedor jetty con el siguiente comando:

```
docker run -d -p 8080:8080 plantuml/plantuml-server:jetty
```

Con la ejecución de este comando, el servidor está arriba y disponible, como indica su repositorio, se consultan los servicios api para la generación de diagramas de manera local en http://localhost:8080/. Cuando se utiliza a través del Docker-file,

se accede directamente desde el nombre de la imagen creada para utilizar el servicio api expuesto.

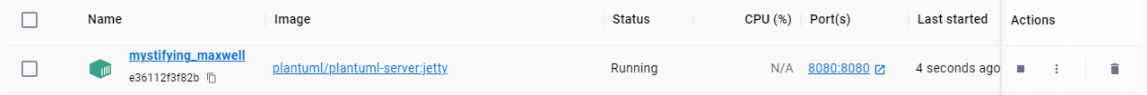


Figura 49. Servidor PlantUML levantado en Docker (Elaboración propia)

Backend en Python para la conexión con OpenAi y el bot de Telegram. Esta es la estructura del servicio.

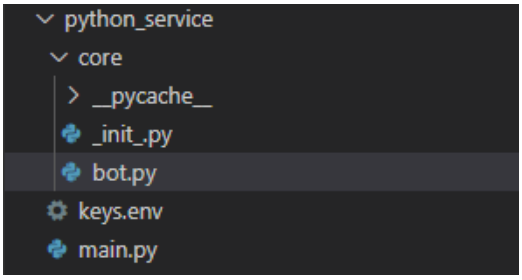


Figura 50. Estructura del proyecto de integración (Elaboración propia)

El archivo keys.env se encuentran las variables de entorno, el token de OpenAI, el token de telegram, el nombre del modelo de Fine-Tunned y el número de tokens máximos para la respuesta.

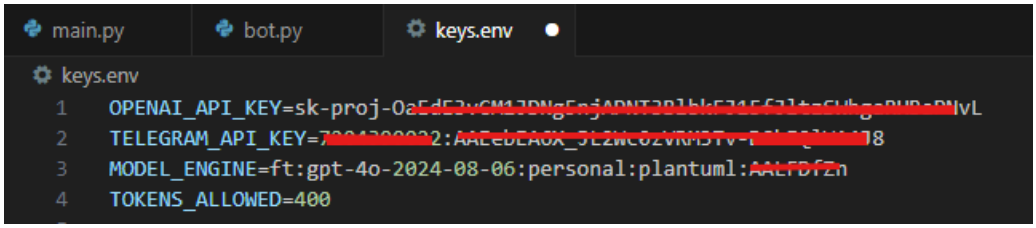


Figura 51. API Keys de OpenAI, Telegram y modelo entrenado (Elaboración propia)

En el archivo main.py se importa la clase ChatBotMaker para inicializarla, enviando la referencia del archivo keys.env:

```
from core.bot import ChatBotMaker

if __name__ == '__main__':
    environment_file_path = r'keys.env'
    my_bot = ChatBotMaker(environment_file_path)
    my_bot.run()
```

En la inicialización de la clase CharBotMaker del archivo bot.py se carga en variables locales las variables de entorno del archivo keys.env:

```
class ChatBotMaker:
    def __init__(self, environment_file):
        load_dotenv(environment_file)
        self.api_key = os.getenv("OPENAI_API_KEY")
        self.token = os.getenv("TELEGRAM_API_KEY")
        self.model_engine = os.getenv("MODEL_ENGINE")
        self.tokens_allowed = os.getenv("TOKENS_ALLOWED")

        # Check all the environment variables
        if not all([self.api_key, self.token,
            self.model_engine]):
            raise ValueError("Some variables are not readed
                        correctly.")
```

El método inicial se mantiene en ejecución para escuchar las peticiones del ChatBot de telegram, después envía la petición a openai, obtiene una respuesta que es procesada en PlantUML para devolver la imagen del uml requerido como respuesta al bot.

```
class ChatBotMaker:
def run(self):

        print("Starting bot...")
        offset = 0
        while True:
            # Listen new messages
            updates = self.get_updates(offset)
            if updates is not None and updates:
                for update in updates:
                    offset = update["update_id"] + 1
                    message = update.get("message", {})

                    # Check the message and the chat_id
                        to continue
                    if "chat" in message and "text" in message:
                        chat_id = update["message"]["chat"]['id']
                        user_message = update["message"]["text"]
                        print(f"Received message:
                              {user_message}")

                        # Get a response from ChatGPT
                        GPT =self.get_openai_response(
                              self.create_openai_message
                              (user_message))
                        print(f"Answer generated: {GPT}")

                        script = self.extract_plantuml_code(GPT)
                        if 'Error' in script:
                            url = script:
                        else:
                            url=self.generate_plantuml_url(
```

```
                                script)

                            # Send the response to the bot
                            self.send_messages(chat_id, url)
                        else:
                            print("No message or chat_id founded
                                  in the update.")
                else:
                    print("No new updates or empty updates
                          received.")
                    time.sleep(1)

                time.sleep(0.5)
```

Esta es la función para obtener los mensajes más recientes del Bot de telegram, el parámetro “offset” se utiliza para indicar el identificador del último mensaje recibido por el bot. Este parámetro se usa en el método “get_updates” para obtener solo los mensajes nuevos que han llegado desde el último mensaje procesado por el bot.

```
class ChatBotMaker:
 def get_updates(self, offset: int):
        url = f"https://api.telegram.org/bot{self.token}
              /getUpdates"
        params = {"timeout": 100, "offset": offset}
        response = requests.get(url, params=params)

        # Check if 'result' is in JSON response
        if response.status_code == 200:
            json_response = response.json()
            if "result" in json_response:
                return json_response["result"]
            else:
                return []  # Return empty if 'result'
                           is not on the response
        else:
            return []  # Return empty if something
                       was not success
```

Este método crea un mensaje en el formato requerido por la API de OpenAI del tipo “user_message”. El cual contiene el rol y el contenido de cada parte.

```
class ChatBotMaker:
 def create_openai_message(self, user_message):
        return [
            {
                "role": "system",
                "content": "Tu eres un asistente de diagramación
                  que devuelve el script de plantUML"
            },
            {
                "role": "user",
                "content": user_message
            }
```

```
        ]
```

En este método se envia un “prompt” de entrada y genera una respuesta, utilizando el modelo de ChatGPT fine-tuned, y controla los errores en caso de no poder completarlo:

```
class ChatBotMaker:
 def get_openai_response(self, prompt):
        try:
            client = OpenAI(api_key=self.api_key)
            response = client.chat.completions.create(
                model=self.model_engine,
                messages=prompt,
                max_tokens= int(self.tokens_allowed),
                n=1,
                temperature=0.5
            )
            print (f"respuesta {response}")
            return
                  response.choices[0].message.content.replace('sk
                  inparam monochrome true\n','').strip()
        except OpenAI.APIError as e:
            # Retry or sing in again
            print(f"OpenAI Api return an error: {e}")
            pass  # Aprove
        except OpenAI.APIConnectionError as e:
            # Conexión error
            print(f"Error connecting to OpenAI API: {e}")
            pass
        except OpenAI.RateLimitError as e:
            # Error with the limit quota
            print(f"The request of OpenAI API exceeded the limit
                  of frequency: {e}")
            pass

        return "An Error happens :("
```

Extrae el script que requiere plantUML según su definición con los delimitadores @startuml y @enduml. Devuelve error si alguno de los delimitadores no está presente, esto ocurre cuando el número de tokens configurado no fue suficiente para cumplir con el script.

```
class ChatBotMaker:
    def extract_plantuml_code(self, text):
        # Regular expression to find @startuml and @enduml
        pattern = r'@startuml(.*?)@enduml'

        # Search coincidences in the text
        match = re.search(pattern, text, re.DOTALL)

        if match:
            # Add again @startuml and @enduml
            return f"@startuml{match.group(1)}@enduml"
        else:
```

```
            return " Error - The response connot load correctly,
                     probably by the token number allowed"
```

Codifica el script obtenido con el formato utf-8 para español, llamando al servicio local desplegado de plantUML.

```
class ChatBotMaker:
    def encode_plantuml(self, plantuml_text):
        connection = http.client.HTTPConnection("localhost",
                     8080)
        encoded_text = plantuml_text.encode("utf-8")
        headers = {
        'Content-Type': 'text/plain; charset=utf-8'
        }
        connection.request("POST", "/coder", encoded_text,
                           headers)
        response = connection.getresponse()
        data = response.read()
        return data.decode("utf-8")
```

Descarga la imagen del diagrama UML solicitado, enviando la codificación correspondiente.

```
class ChatBotMaker:
    def generate_plantuml_url(self, plantuml_text):
        encoded = self.encode_plantuml(plantuml_text)
        print(encoded)
        return f"http://www.plantuml.com/plantuml/png/{encoded}"
```

Se envia un mensaje del tipo respuesta del BOT, al Usuario de Telegram con la imagen del diagrama uml obtenido o el error correspondiente.

```
class ChatBotMaker:
    def send_messages(self, chat_id, text: str):
        if text.startswith("http"):  # If the text is a URL
            url = f"https://api.telegram.org/bot{self.token}
                  /sendPhoto"
            params = {"chat_id": chat_id, "photo": text}
        else:  # If the text is a simple string
            url = f"https://api.telegram.org/bot{self.token}
                  /sendMessage"
            params = {"chat_id": chat_id, "text": text}

        response = requests.post(url, params=params)
        return response
```

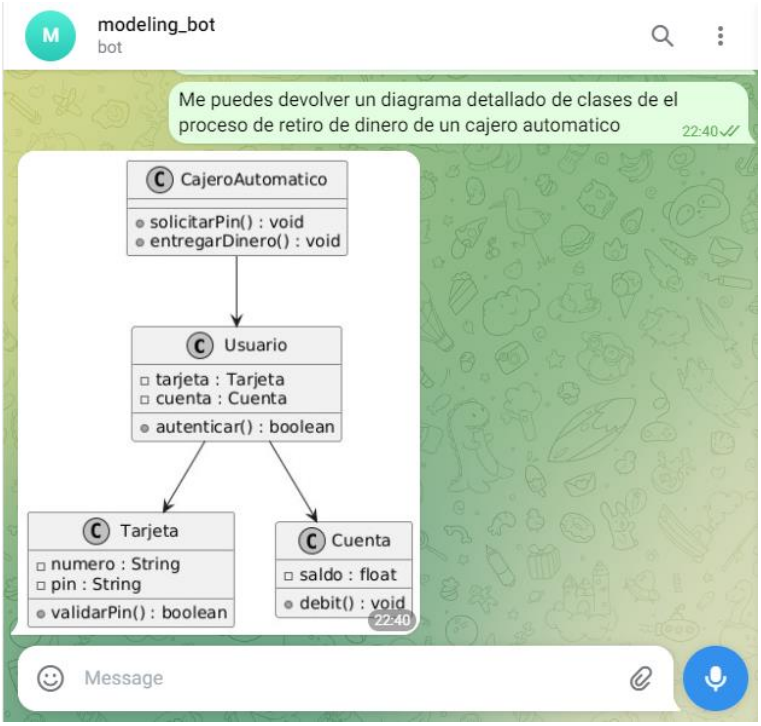


Figura 52. Ejemplo de funcionamiento de la solución (Elaboración propia)

La API de integración desarrollada demuestra la viabilidad de conectar el modelo GPT-4 de OpenAI con la API de PlantUML y el chatbot de Telegram. Las pruebas realizadas confirmaron el correcto funcionamiento de esta implementación y la capacidad del sistema para generar diagramas UML de manera real a partir de descripciones en lenguaje natural. Esta solución modular facilita futuras expansiones o mejoras en los servicios integrados.

3.7. Evaluación del Comportamiento del Modelo a Través de la Interfaz

El objetivo de esta evaluación es analizar la precisión, consistencia y rendimiento del modelo GPT-4 de OpenAI en la generación de diagramas UML a través de una integración que conecta el chatbot de Telegram con la API de PlantUML. La evaluación incluye pruebas en distintos casos de uso para evaluar la adaptabilidad del sistema a una variedad de escenarios y su eficacia en la generación automática de diagramas UML complejos.

1. **Pruebas de Precisión**

Estas pruebas se enfocan en verificar la correspondencia entre las descripciones en lenguaje natural proporcionadas por el usuario y los diagramas UML generados automáticamente. La precisión se mide en diferentes tipos de diagramas UML, como clases, actividad, secuencias y casos de uso, para asegurar una interpretación correcta de los conceptos de UML.

- **Métrica de Precisión**: Evalúa la precisión en la interpretación de términos técnicos de UML, incluyendo asociaciones, herencias y multiplicidades. Cada diagrama generado fue comparado con un diagrama de referencia, creado manualmente, para confirmar la exactitud de las estructuras y las interconexiones. Se observa que el modelo responde de manera satisfactoria cuando se especifican detalles en la petición, como la necesidad de incluir herencia o patrones de desarrollo específicos.

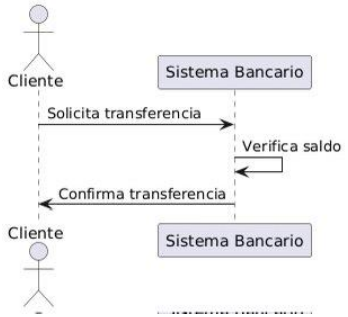
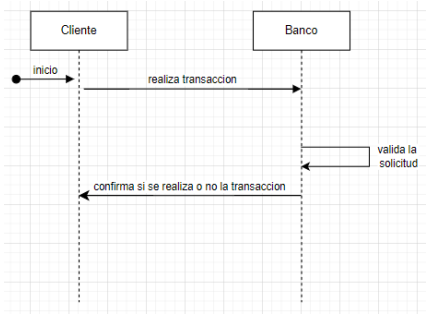
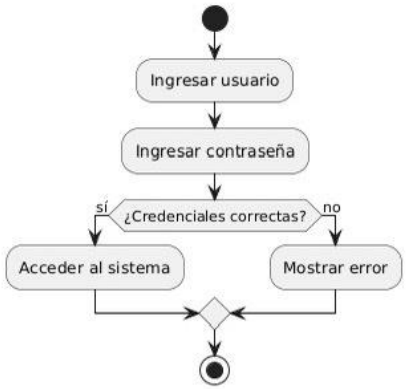
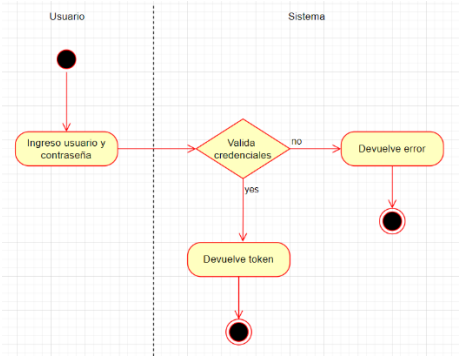
  Estos resultados sugieren que el sistema maneja adecuadamente peticiones de baja y mediana complejidad, generando diagramas entendibles y legibles. Además, esta precisión facilita la comprensión por parte de los usuarios, quienes pueden visualizar diagramas claros y bien estructurados, consistentes con el diseño solicitado.

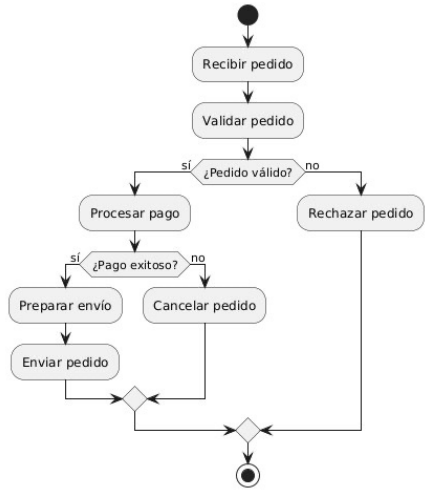
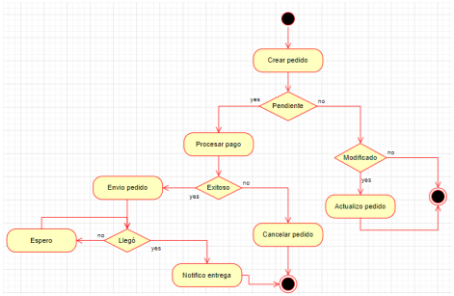
2. **Pruebas de Rendimiento**

La evaluación de rendimiento se centró en medir los tiempos de respuesta en cada paso del proceso, desde que el usuario envía la solicitud en Telegram hasta que recibe el diagrama generado por PlantUML. Estas pruebas son esenciales para asegurar que el sistema pueda manejar la generación de diagramas de manera eficiente, incluso en casos de mayor complejidad.

- **Métrica de Rendimiento**: Se midió el tiempo total de respuesta desde la solicitud hasta la entrega del diagrama generado automáticamente, comparándolo con el tiempo que lleva realizar los mismos diagramas de manera manual. Este análisis busca verificar que el sistema cumple con uno de los objetivos clave: optimizar los tiempos en el ciclo de vida del desarrollo de software mediante la generación rápida de diagramas UML.

La comparativa demostró una eficiencia notable del sistema automático, particularmente en tareas complejas, donde logra superar al método manual en rapidez sin comprometer la claridad y calidad de los diagramas. Esta mejora en tiempos de respuesta contribuye a una mayor agilidad en el proceso de documentación y diseño de software, permitiendo a los equipos concentrarse en otras etapas críticas del desarrollo.

Solicitud	Tiempo (s)	Desarrollado por IA	tokens	Desarrollado por persona	Tiempo Manual (seg)
Diagrama de secuencia para una transacción bancaria	12	 *Figura 53. Respuesta de solicitud para diagrama Uml1*	103	 *Figura 54. Respuesta de solicitud manual para diagrama Uml1*	480
Diagrama de actividad para el proceso de inicio de sesión	7	 *Figura 55. Respuesta de solicitud para diagrama Uml2*	94	 *Figura 56. Respuesta de solicitud manual para diagrama Uml2*	300

¿Cómo puedo representar un diagrama de actividad para el ciclo de vida de un pedido?	9	 *Figura 57. Respuesta de solicitud para diagrama Uml3*	109	 *Figura 58. Respuesta de solicitud manual para diagrama Uml3*	900

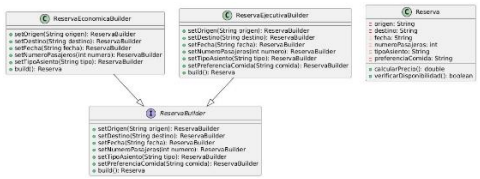
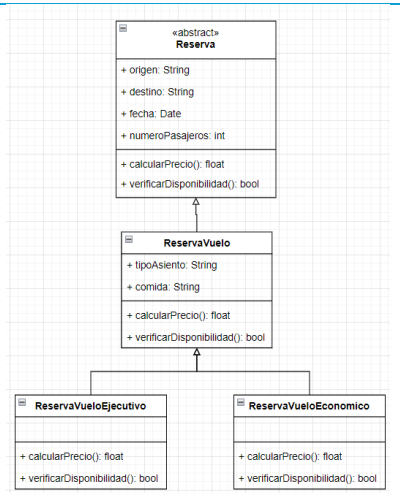
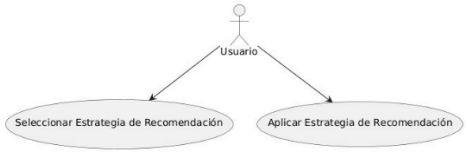
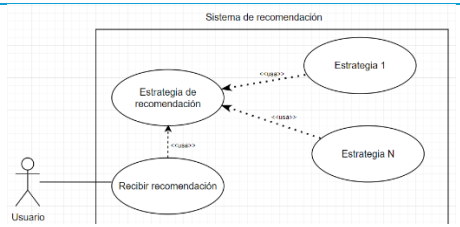
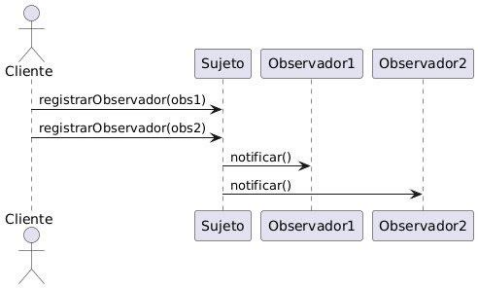
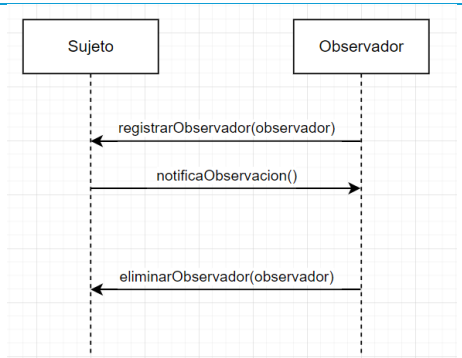
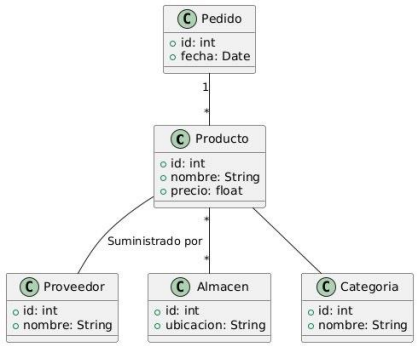
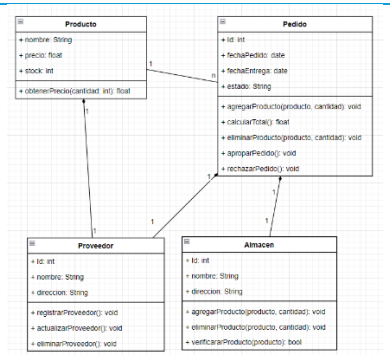
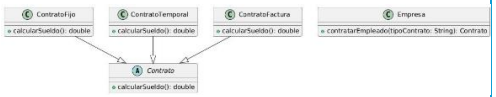
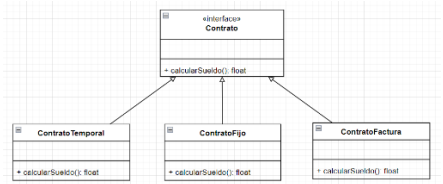
Crea un diagrama de clases de un sistema de reserva de vuelos, se requiere una clase compleja para manejar los detalles de la reserva de un vuelo. Los atributos relacionados con la reserva, cómo el origen, destino, fecha y número de pasajeros tienen similitudes, pero difieren en cuanto al cálculo de los precios y disponibilidad. Algunos atributos cómo el tipo de asiento o la preferencia de comida pueden ser opcionales en algunas reservas, pero obligatorios en otras.	15	 *Figura 59. Respuesta de solicitud para diagrama Uml4*	462	 *Figura 60. Respuesta de solicitud manual para diagrama Uml4*	360
Diagrama de casos de uso para el patrón Strategy en un sistema de recomendación	5	 *Figura 61. Respuesta de solicitud para diagrama Uml5*	114	 *Figura 62. Respuesta de solicitud manual para diagrama Uml5*	300

Diagrama de secuencia para el patrón Observer	9	 *Figura 63. Respuesta de solicitud para diagrama Uml6*	120	 *Figura 64. Respuesta de solicitud manual para diagrama Uml6*	300
Diagrama de clases complejo para un sistema de inventario, que incluya Producto, Proveedor, Pedido, Almacén y Categoría de productos.	11	 *Figura 65. Respuesta de solicitud para diagrama Uml7*	189	 *Figura 66. Respuesta de solicitud manual para diagrama Uml7*	900

Supongamos que en una empresa se contrata a los empleados mediante 3 tipos de contratos: Fijo, Temporal y Mediante Factura. Debe existir una sola clase Contrato que permita calcular el sueldo, pero cada una de las subclases de Contrato pueden especificar su funcionamiento y el cálculo del sueldo es diferente en cada uno de los contratos. No se puede saber de antemano la clase que se instanciará, ya que dependerá de un parámetro ingresado por el cliente.	9	 *Figura 67. Respuesta de solicitud para diagrama Uml8*	239	 *Figura 68. Respuesta de solicitud manual para diagrama Uml8*	600

3. **Evaluación de Casos de Uso Complejos**

El modelo se probó en escenarios de mayor complejidad, donde se utilizaban múltiples relaciones entre clases y diagramas con diversas estructuras. El objetivo era verificar si el sistema podía mantener la precisión en la generación de diagramas más avanzados.

- **Manejo de Errores**: En escenarios donde el modelo superó el número de tokens configurados, el modelo no presenta respuesta del diagrama, sino en su defecto indica que no tiene suficiente número de tokens configurados para proporcionar una respuesta.

Figura 69. Mensaje de error de la solución cuando superó el número de tokens (Elaboración propia)

4. **Evaluación Comparativa**

Luego de todo el proceso que ha conllevado la iteración y entrenamiento del modelo, y considerando que se tienen 5 versiones disponibles en donde se

evidencia este proceso de entrenamiento. Utilizaremos la solución implementada para las pruebas de comparación, obteniendo así los siguientes resultados.

Modelo V1 (gpt-3.5-turbo):

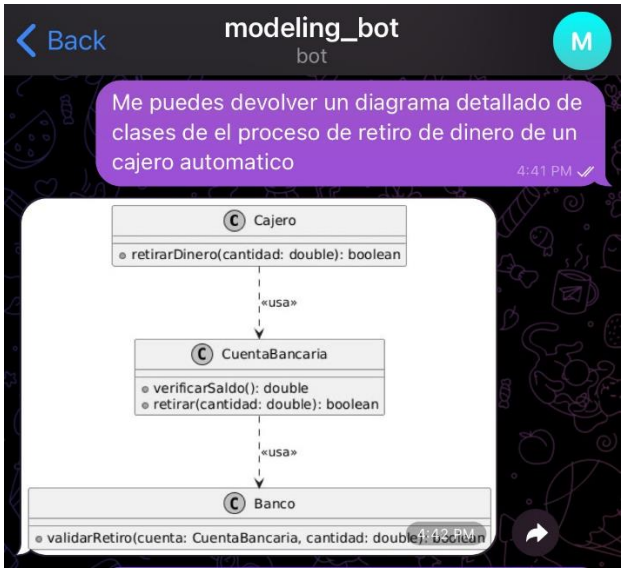


Figura 70. Ejemplo de solicitud para diagrama Uml con la Versión del Modelo 1 (Elaboración propia)

- **Enfoque:** El modelo devuelve un diagrama abstracto que se centra en las interacciones lógicas entre un Cajero, una CuentaBancaria, y un Banco.
- **Fortalezas:** El diagrama cubre las relaciones fundamentales del proceso (retiro, verificación de saldo).
- **Limitaciones:** Falta detalle sobre la participación del cliente o de los dispositivos físicos involucrados en el proceso.
- **Conclusión:** Esta versión ofrece una representación funcional básica pero incompleta del proceso de retiro de dinero, enfocándose solo en las entidades bancarias.

1. **Modelo V2 (gpt-3.5-turbo):**

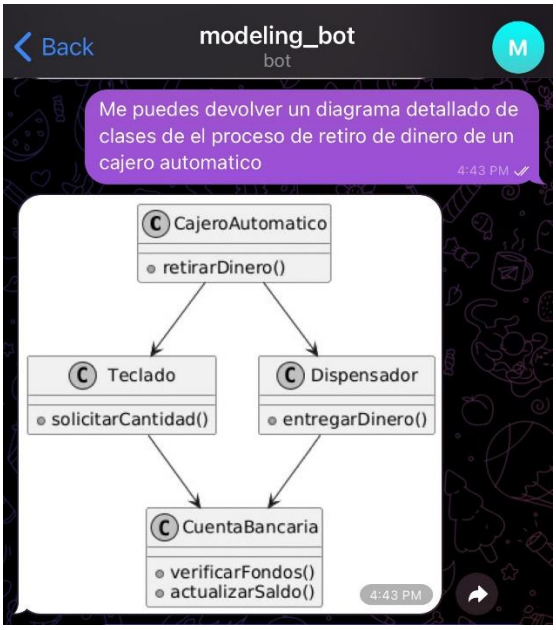


Figura 71. Ejemplo de solicitud para diagrama Uml con la Versión del Modelo 2 (Elaboración propia)

- **Enfoque:** El modelo amplía la cobertura al incluir componentes físicos como el Teclado y el Dispensador.
- **Fortalezas:** Añade más detalle sobre los componentes físicos del cajero, lo cual enriquece la representación del flujo del proceso desde la perspectiva de hardware.
- **Limitaciones:** No hay inclusión directa del usuario como actor, ni mención de un mecanismo de autenticación como la tarjeta o el PIN.
- **Conclusión:** Aunque el diagrama es más detallado, sigue careciendo de la interacción directa del usuario y el flujo de autenticación.

2. **Modelo V3 (gpt-3.5-turbo):**

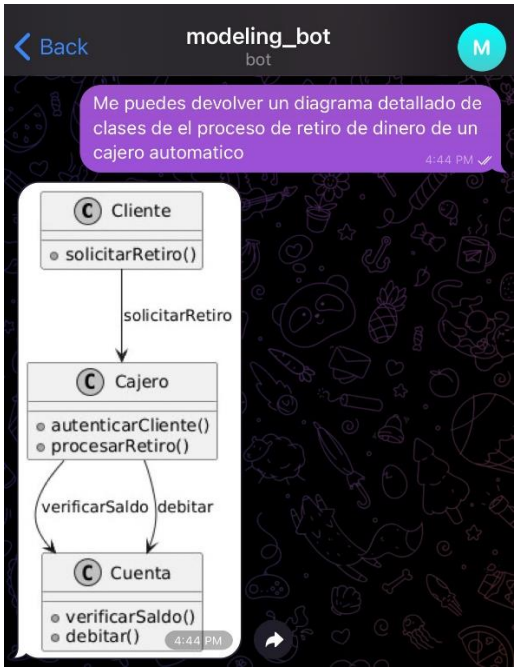


Figura 72. Ejemplo de solicitud para diagrama Uml con la Versión del Modelo 3 (Elaboración propia)

- **Enfoque:** En esta versión, el bot introduce al Cliente como actor, lo cual agrega una capa significativa al proceso de retiro.
- **Fortalezas:** Mejora al incluir métodos de interacción del cliente como solicitarRetiro() y la verificación del saldo, aunque todavía no se maneja la autenticación con tarjeta o PIN.
- **Limitaciones:** Aún no se aborda la seguridad con tarjeta ni el proceso de validación del PIN, lo que limita la precisión del diagrama en comparación con versiones futuras.
- **Conclusión:** Es un progreso notable en cuanto a la interacción del cliente, pero aún faltan detalles cruciales del proceso, como el uso de la tarjeta y PIN.

3. **Modelo V4 (gpt-4o):**

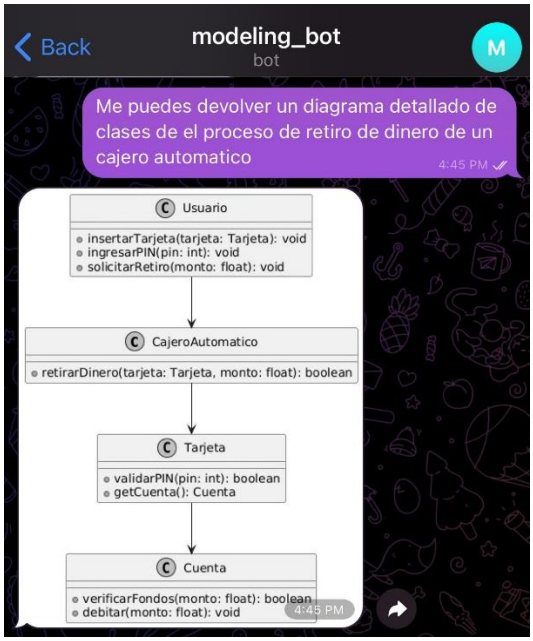


Figura 73. Ejemplo de solicitud para diagrama Uml con la Versión del Modelo 4 (Elaboración propia)

- **Enfoque:** Se introduce la clase Tarjeta y el proceso de autenticación mediante PIN, mostrando una interacción más completa y cercana a la realidad.
- **Fortalezas:** Este diagrama cubre el flujo de interacción del Usuario, el uso de Tarjeta y el PIN, lo que mejora notablemente el nivel de detalle y precisión del modelo.
- **Limitaciones:** Aunque es bastante detallado, los tipos de datos de los métodos no están especificados, lo que podría dificultar la implementación directa.
- **Conclusión:** Esta versión es mucho más detallada y completa en cuanto al flujo del proceso, mostrando cómo se valida la tarjeta y se verifica el saldo, aunque aún se pueden agregar más detalles técnicos.

4. **Modelo V5 (gpt-4o):**

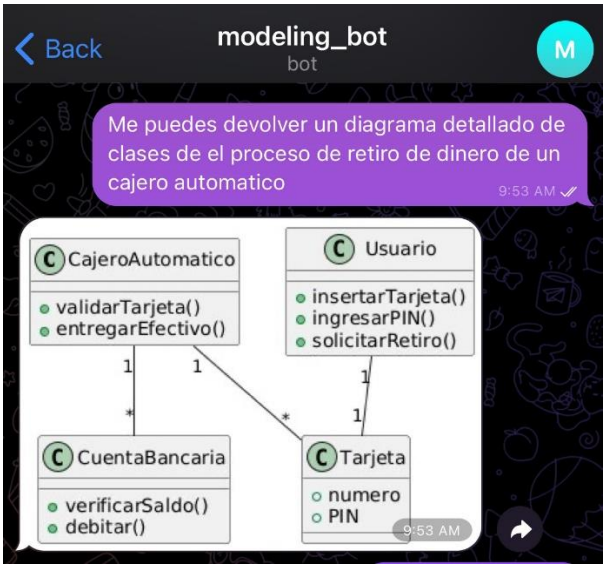


Figura 74. Ejemplo de solicitud para diagrama Uml con la Versión del Modelo 5 (Elaboración propia)

- **Enfoque:** Es la versión más completa, detallando cada clase con sus métodos y tipos de datos específicos. Incluye claramente al Usuario, la Tarjeta, y la interacción con el CajeroAutomático.
- **Fortalezas:** Esta es la versión más robusta en términos de implementación técnica. Especifica los parámetros de los métodos (tarjeta: Tarjeta, monto: float, pin: int) y los tipos de retorno (boolean, void), lo que ofrece una perspectiva más exacta del flujo de trabajo.
- **Limitaciones:** Ninguna significativa, es la versión más detallada y precisa del proceso de retiro de dinero.
- **Conclusión:** Esta versión refleja el resultado de un entrenamiento avanzado del modelo, capturando todos los elementos clave del proceso con detalles técnicos específicos, lo que la convierte en la mejor representación.

Conclusión

El modelo se comparó con sistemas convencionales de generación de diagramas para evaluar si el enfoque basado en lenguaje natural podía ofrecer mejoras significativas en cuanto a accesibilidad y facilidad de uso. Esto permitió analizar la viabilidad del sistema en el contexto del ciclo de vida de desarrollo de software.

Las pruebas realizadas confirman que el modelo de IA, integrado a través de la API, tiene la capacidad de crear diagramas UML con un alto grado de precisión y con

tiempos aceptables. Las mejoras identificadas durante las pruebas permitirán optimizar el rendimiento y la adaptabilidad del sistema a casos de uso más complejos en el futuro.

4. Resultados y discusión

4.1 Evaluación y mejora a través de las versiones de modelos fine - tuned

A lo largo de las versiones, el bot ha mejorado notablemente en términos de:

- **Complejidad:** Desde una versión inicial con pocas clases hasta un desglose más detallado de cada interacción.
- **Precisión:** Cada iteración añade más precisión al incluir el uso de tarjeta, PIN, y la verificación del saldo.
- **Flujo Completo:** La evolución del modelo refleja un proceso mucho más realista, pasando de interacciones lógicas básicas a un flujo complejo con actores, dispositivos físicos y autenticación.
- **Detallado Técnico:** La Versión 5 es la más avanzada, ya que especifica claramente los tipos de datos y retorno de los métodos, lo cual es crucial para una implementación precisa.

El proceso de entrenamiento del bot ha mejorado significativamente desde la Versión 1 hasta la Versión 5. Cada versión aporta un nivel adicional de detalle, hasta alcanzar un modelo completamente funcional y detallado en la versión final. Esta evolución demuestra cómo el bot ha aprendido a manejar casos más complejos y específicos, pasando de respuestas abstractas a una representación detallada y técnica del proceso de retiro de dinero en un cajero automático, incluso cuando las instrucciones son justas y limitadas.

Aun cuando se evidencia el avance y la mejora en la versión 5, es importante reconocer que los modelos de openAI crecen a una velocidad exponencial, lo cual incita a siempre estar refinando y mejorando el modelo para el objetivo planteado. Entonces, como resultado se reconoce que el proceso de fine-tuning consiste en

ejecutar un proceso iterativo para evidenciar mejorías, pero se deben realizar cientos de pruebas más exactas hasta llegar a un grado de complejidad que satisfaga el uso de este trabajo.

5. Conclusiones

La investigación reveló que los modelos de lenguaje de OpenAI poseen capacidades avanzadas en procesamiento de lenguaje natural (NLP), permitiendo interpretar descripciones en lenguaje natural para crear contenido estructurado, como diagramas UML. Esta característica potencia la automatización en la documentación y diseño de software, alcanzando una interpretación precisa de los requerimientos iniciales de los usuarios.

La integración de OpenAI con un chatbot de Telegram facilita una interfaz intuitiva y accesible, lo que abre una alternativa para que los usuarios se conecten con el sistema en tiempo real, sin necesidad de tener conocimientos profundos en modelado UML. Esta implementación demuestra la viabilidad de incorporar IA en procesos de diseño iterativo, optimizando la eficiencia en la generación de diagramas.

La investigación sobre PlantUML evidenció que esta herramienta proporciona una estructura flexible y programática para la generación de diagramas, ideal para integrarse con un modelo de IA como el de OpenAI. A través de su API, es posible automatizar la creación de diagramas UML a partir de descripciones textuales, agilizando la actualización de diagramas ante ajustes en los requisitos del proyecto.

La implementación de PlantUML en combinación con un modelo de lenguaje permite aprovechar la API para producir diagramas de calidad, de manera eficiente y con un elevado nivel de precisión. Esta capacidad de traducción de texto a imagen representa una ventaja en entornos de desarrollo ágil, donde los cambios rápidos en los requisitos son comunes.

El entrenamiento iterativo del modelo de OpenAI demostró ser fundamental para mejorar su capacidad de comprensión en contextos de generación de diagramas UML. A través de múltiples ciclos de ajuste y pruebas, se logró afinar el modelo para

que interprete con mayor precisión las descripciones en lenguaje natural, lo cual reduce errores y mejora la consistencia en los resultados generados.

Las pruebas continuas en el proceso de entrenamiento permitieron identificar y corregir limitaciones específicas del modelo, incrementando su adaptabilidad y precisión. Esta iteración constante validó que el enfoque de fine-tuning es efectivo para personalizar el modelo y adecuarlo a necesidades específicas de diseño UML.

La integración de una API entre OpenAI y PlantUML fue exitosa, creando un flujo de trabajo en el que el chatbot de Telegram actúa como intermediario, facilitando la comunicación y solicitud de diagramas UML de forma sencilla. Esta implementación confirma que los microservicios pueden coordinar efectivamente las respuestas de OpenAI y PlantUML, generando una solución integral y escalable.

Las pruebas realizadas con la API integrada mostraron que la arquitectura basada en microservicios facilita la interoperabilidad entre modelos de IA y herramientas de modelado, permitiendo una automatización fluida de la generación de diagramas UML. Esto confirma que los sistemas basados en API pueden optimizar procesos complejos en el desarrollo de software.

La evaluación final del modelo de IA, a través del chatbot de Telegram, indicó que la solución no solo genera diagramas UML precisos, sino que también mejora la accesibilidad y reduce tiempos en el proceso de diseño. Los usuarios pueden describir sus necesidades en lenguaje natural y recibir diagramas ajustados en tiempo real, lo cual optimiza la experiencia y promueve la eficiencia en el ciclo de vida de desarrollo de software.

Las pruebas en diferentes escenarios de uso permitieron verificar que el modelo mantiene la precisión y adaptabilidad ante casos complejos, logrando una generación de diagramas consistente y relevante. Esto confirma que la solución es robusta y adaptable a diferentes contextos de diseño UML, brindando un recurso efectivo para desarrolladores y analistas.

Referencias

Accelerate State of DevOps 2021 | Google Cloud. (n.d.). https://cloud.google.com/resources/state-of-devops?hl=es

Admin. (2023). Pasa al siguiente nivel de automatización del ciclo de vida software. https://luceit.com/es/blog/mejora-de-procesos/pasa-al-siguiente-nivel-de-automatizacion-del-ciclo-de-vida-software/

Aguilera, C. (2024). API testing: guía práctica introductoria. https://es.abstracta.us/blog/api-testing-guia-practica/

Álvaro, C. P. (2017). Desarrollo de un juego de preguntas basado en un Bot enfocado a la educación en salud. https://academica-e.unavarra.es/entities/publication/68eeb08c-26c4-4f9b-9107-735b33c1f435

Ambler, S. (2002). Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process. In John WileySons.

Atlassian. (n.d.). Arquitectura de microservicios | Atlassian. https://www.atlassian.com/es/microservices/microservices-architecture

Avuthu, S. S. R. (2023). Building a Telegram Bot with AWS API Gateway, AWS Lambda, and Postman. https://www.linkedin.com/pulse/building-telegram-bot-aws-api-gateway-lambda-postman-avuthu

Belandria, J. (2023). Automatización de Procesos: todo lo que necesitas saber. https://www.gb-advisors.com/es/automatizacion-de-procesos/

Benavides, L., Goytia, E., & Ramirez, L. (2023). El uso de la inteligencia artificial en un entorno académico. Ciencia Nicolaita. https://doi.org/10.35830/cn.vi89.721

Brown, S. (2020). Modelling software architecture with PlantUML. https://dev.to/simonbrown/modelling-software-architecture-with-plantuml-56fc

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J.,

Winter, C., … Amodei, D. (2020). Language models are few-shot learners. Advances in Neural Information Processing Systems, 2020-December.

Comprendiendo la arquitectura de los chatbots. (n.d.). https://www.chatcompose.com/arquitectura.html

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference, 1.

Diaz, B., & Diaz, B. (2023). #13 El ABC del procesamiento de lenguaje natural. https://impulsatek.com/13-el-abc-del-procesamiento-de-lenguaje-natural/

Fuentes, L., & Vallecillo, A. (2004). Una introducción a los perfiles UML. Novática, 168, 6–11.

Goodfellow Ian, Bengio Yoshua, C. A. (2016). Deep Learning - Ian Goodfellow, Yoshua Bengio, Aaron Courville - Google Books. In MIT Press.

Grady, B., Rumbaugh, J., & Jacobson, I. (2005). Unified Modeling Language User Guide. ResearchGate.

Hussain, M. S. (2023). Evolution of GPT series: The GPT revolution from 1 to 4 trillion | Data Science Dojo. https://datasciencedojo.com/blog/evolution-of-gpt-series/

IBM Watson Assistant, Telegram and Python. (n.d.). https://stackoverflow.com/questions/53757816/ibm-watson-assistant-telegram-and-python

Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. In Science (Vol. 349, Issue 6245). https://doi.org/10.1126/science.aaa8415

Kaplan, A., & Haenlein, M. (2019). Siri, Siri, in my hand: Who's the fairest in the land? On the interpretations, illustrations, and implications of artificial intelligence. In Business Horizons (Vol. 62, Issue 1). https://doi.org/10.1016/j.bushor.2018.08.004

Keselj, V. (2009). Speech and Language Processing (second edition) Daniel Jurafsky and James H. Martin (Stanford University and University of Colorado at Boulder) Pearson Prentice Hall, 2009, xxxi+988 pp; hardbound, ISBN 978-0-13-187321-6, $115.00. Computational Linguistics, 35(3). https://doi.org/10.1162/coli.b09-001

LeCun, Y., Bengio, Y., Hinton, G. (2015). Deep learning. nature 521 (7553): 436. Nature, 521.

Lee, L. (2000). Foundations of Statistical Natural Language Processing Christopher D. Manning and Hinrich Schütze (Stanford University and Xerox Palo Alto Research Center) Cambridge, MA: The MIT Press, 1999, xxxvii + 680 pp; hardbound, ISBN 0-262-13360-1, $60.00 . Computational Linguistics, 26(2). https://doi.org/10.1162/coli.2000.26.2.277

Murphy, K. P. (2012). Machine Learning - A Probabilistic Perspective - Table-of-Contents. The MIT Press.

OpenAI. (n.d.). OpenAI Website.

PlantUML Website. (n.d.).

Priore, P., De La Fuente, D., Puente, J., & Gómez, A. (2001). Secuenciación dinámica de sistemas de fabricación flexible mediante aprendizaje automático: Análisis de los principales sistemas de secuenciación existentes. Questiio, 25(3).

Radford Alec, Wu Jeffrey, Child Rewon, Luan David, Amodei Dario, & Sutskever Ilya. (2019). Language Models are Unsupervised Multitask Learners | Enhanced Reader. OpenAI Blog, 1(8).

Russell, S., & Norvig, P. (2021). Artificial Intelligence: A Modern Approach (4th ed.). Pearson Education.

TRPlane. (2021). Comprender el ciclo de Cida de Desarrollo de Software (SLDC). https://www.trplane.com/comprender-el-ciclo-de-cida-de-desarrollo-de-software-sldc/

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. Advances in Neural Information Processing Systems, 2017-December.

Wolchover, N. (2020). New Theory Cracks Open the Black Box of Deep Learning. In Alice and Bob Meet the Wall of Fire. https://doi.org/10.7551/mitpress/11909.003.0037

Zaremba, W., Brockman, G., & OpenAI. (2021). OpenAI Codex. Https://Openai.Com/Blog/Openai-Codex.