ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

Facultad de Ingeniería en Electricidad y Computación

“DISEÑO DE DIAGRAMAS DE COMPONENTES DE SOFTWARE, UTILIZANDO UN GRAN MODELO DE LENGUAJE Y APLICANDO TÉCNICAS DE OPTIMIZACIÓN PARA LOGRAR RESULTADOS CONTEXTUALMENTE RELEVANTES”

TESIS DE GRADO

Previa a la obtención del Título de:

MAGISTER EN SISTEMAS DE INFORMACIÓN GERENCIAL

Presentada por:

JUAN FRANCISCO ROMERO AGUILAR

GUAYAQUIL – ECUADOR

AÑO 2024

AGRADECIMIENTO

Expreso mi agradecimiento a Dios por darme la salud y la fortaleza necesarias para llevar a cabo este trabajo. A mi empresa y a mis jefas, por su constante apoyo en la consecución de esta meta. Y a la ESPOL, así como al Mgs. Lenin Freire, por brindarme la invaluable oportunidad de cursar esta maestría en su prestigiosa institución.

Ing. Juan Romero Aguilar

DEDICATORIA

Dedico este trabajo a mis padres, pilares fundamentales de lo que soy hoy; a mis hermanas, ejemplo constante de superación; a mi esposa, mi apoyo incondicional; y a mis hijas, el motor que me impulsa a seguir cada día.

Ing. Juan Romero Aguilar

DECLARACIÓN EXPRESA

Yo Juan Francisco Romero Aguilar acuerdo y reconozco que: La titularidad de los derechos patrimoniales de autor (derechos de autor) del proyecto de graduación corresponderá al autor, sin perjuicio de lo cual la ESPOL recibe en este acto una licencia gratuita de plazo indefinido para el uso no comercial y comercial de la obra con facultad de sublicenciar, incluyendo la autorización para su divulgación, así como para la creación y uso de obras derivadas. En el caso de usos comerciales se respetará el porcentaje de participación en beneficios que corresponda a favor del autor. El estudiante deberá procurar en cualquier caso de cesión de sus derechos patrimoniales incluir una cláusula en la cesión que proteja la vigencia de la licencia aquí concedida a la ESPOL.

La titularidad total y exclusiva sobre los derechos patrimoniales de patente de invención, modelo de utilidad, diseño industrial, secreto industrial, secreto empresarial, derechos patrimoniales de autor sobre software o información no divulgada que corresponda o pueda corresponder respecto de cualquier investigación, desarrollo tecnológico o invención realizada por mí durante el desarrollo del proyecto de graduación, pertenecerán de forma total, exclusiva e indivisible a la ESPOL, sin perjuicio del porcentaje que me corresponda de los beneficios económicos que la ESPOL reciba por la explotación de mi innovación, de ser el caso.

En los casos donde la Oficina de Transferencia de Resultados de Investigación (OTRI) de la ESPOL comunique al autor que existe una

innovación potencialmente patentable sobre los resultados del proyecto de graduación, no se realizará publicación o divulgación alguna, sin la autorización expresa y previa de la ESPOL.

Guayaquil, noviembre del 2024.

Firmado electrónicamente por:
JUAN FRANCISCO
ROMERO AGUILAR

ING. JUAN FRANCISCO ROMERO AGUILAR

EVALUADORES

Firmado electrónicamente por:
LENIN EDUARDO FREIRE COBO

Mgs. Lenin Eduardo Freire Cobo

PROFESOR TUTOR

Firmado electrónicamente por:
OMAR RODOLFO MALDONADO DANIN

Mgs. Omar Maldonado

PROFESOR EVALUADOR

RESUMEN

El diseño arquitectónico es un proceso crítico dentro del ciclo de vida del desarrollo de software, donde la precisión y la claridad en la representación de los componentes juegan un papel esencial para asegurar la calidad y mantenibilidad del sistema. No obstante, este proceso suele ser lento y demandante, lo que puede convertirlo en un cuello de botella en organizaciones donde la rapidez de respuesta es crucial. En este contexto, la automatización de la generación de diagramas de componentes de software se presenta como una solución innovadora que permite a los arquitectos optimizar su flujo de trabajo y enfocarse en actividades de mayor valor añadido.

Este trabajo de investigación se centra en el diseño de una herramienta basada en inteligencia artificial, utilizando un Gran Modelo de Lenguaje (LLM) enriquecido con técnicas de Recuperación Aumentada por Generación (RAG), con el objetivo de generar automáticamente diagramas de componentes a partir de descripciones textuales proporcionadas por el usuario. El enfoque propuesto no solo automatiza una parte fundamental del diseño arquitectónico, sino que también asegura que los diagramas generados sean contextualmente relevantes, al integrar información existente en la organización. De esta forma, la herramienta busca reducir significativamente los tiempos de diseño y minimizar el riesgo de errores u omisiones humanos.

El proceso metodológico de este trabajo incluye una fase de levantamiento de información con el equipo de arquitectura de una empresa del sector de telecomunicaciones, lo que permitió identificar los criterios clave para la generación de diagramas precisos y útiles. A partir de esta información, se diseñó e implementó un prototipo que combina el uso de herramientas como PlantUML para la visualización de diagramas y el procesamiento de datos con un LLM local para asegurar la confidencialidad de la información sensible de la organización.

La evaluación del prototipo se realizó mediante pruebas con arquitectos de software, quienes proporcionaron retroalimentación positiva respecto a la usabilidad, eficiencia y precisión de la herramienta. Los resultados mostraron

una reducción del tiempo de generación de diagramas mejorando la eficiencia del equipo de arquitectura. Sin embargo, también se identificaron algunas limitaciones, como la necesidad de mejorar la precisión en la generación de diagramas más complejos, como los de clases, y la integración con otros sistemas corporativos como los repositorios de control de versiones.

Finalmente, el estudio concluye que el uso de tecnologías avanzadas como los LLMs, junto con técnicas de optimización de resultados como RAG, tiene un alto potencial para transformar el proceso de diseño arquitectónico en entornos empresariales. Las recomendaciones futuras incluyen la implementación de mejoras sugeridas por los usuarios, como la generación de diferentes tipos de diagramas UML y una mayor integración con sistemas existentes. Se espera que este prototipo pueda escalarse para cubrir otras áreas del ciclo de desarrollo de software y convertirse en una herramienta clave dentro de los procesos de ingeniería de software moderna.

ÍNDICE GENERAL

ABREVIATURAS

SPA	Single Page Application
LLM	Large Language Model
RAG	Retrieval-Augmented Generation
JVM	Java Virtual Machine
IDE	Integrated Development Environment
API	Application Programming Interface
UML	Unified Modeling Language
DB	Database
TCP	Transmission Control Protocol
gRPC	Google Remote Procedure Call
AS-IS	Representación de la situación actual
TO-BE	Representación de la situación futura

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

En el contexto actual del desarrollo de software, los arquitectos de software enfrentan grandes desafíos en la creación de diagramas que representen la estructura y los componentes de las soluciones informáticas. La creciente complejidad de los sistemas, junto con la necesidad de mantener la coherencia entre los diagramas y la evolución continua de los proyectos, exige herramientas que permitan a los arquitectos generar estos diagramas de manera eficiente y precisa. Tradicionalmente, estos diagramas se crean manualmente, lo que consume tiempo valioso y es propenso a errores u omisiones.

El presente trabajo busca abordar esta problemática mediante el diseño de una herramienta que, utilizando un Gran Modelo de Lenguaje (LLM) y técnicas de Recuperación Aumentada por Generación (RAG), automatice el proceso de generación de diagramas de componentes de software. Este enfoque no solo reduce el tiempo necesario para producir los diagramas, sino que también mejora la relevancia contextual al basarse en datos ya existentes dentro de la organización.

A lo largo del documento, se describen los pasos seguidos para diseñar, desarrollar y validar un prototipo funcional basado en la propuesta. Se analizan las ventajas de esta herramienta en comparación con los métodos tradicionales y se proponen mejoras para aumentar su funcionalidad en futuros desarrollos. Además, se destaca la importancia de esta solución en entornos corporativos, donde la agilidad y precisión son factores críticos para el éxito del desarrollo de software.

CAPÍTULO I

GENERALIDADES

En el presente capítulo se abordará la oportunidad de mejora identificada en uno de los principales procesos de en una empresa del sector de las telecomunicaciones. Se hará una revisión de los antecedentes y el problema, así como el planteamiento de la solución a dicho problema y cuáles serán los objetivos perseguidos para poder cumplir con la solución propuesta, definiendo una metodología para dicho fin.

1.1 Antecedentes

En la ciudad de Guayaquil, lleva a cabo sus operaciones una empresa de Telecomunicaciones que brinda servicios basados en Internet en el sector corporativo, y como toda empresa vanguardista, tiene como uno de sus pilares fundamentales al departamento de Software Factory, en el cual se llevan a cabo los procesos de diseño, desarrollo y mantenimiento de los sistemas informáticos que soportan la operación. El departamento de Software Factory tiene a su vez una división de arquitectura, en la cual se lleva a cabo el diseño de las soluciones, con el objetivo de satisfacer los requisitos que forman parte de cada uno de los proyectos que le son asignados.

Para cumplir con este objetivo, el área de arquitectura realiza una serie de actividades, las cuales requieren tiempo y esfuerzo, entre las cuales destacan:

- Análisis de requisitos de usuario.
- Análisis de la solución.
- Diseño de diagrama de componentes de software.
- Diseño de diagramas de secuencia.
- Diseño de diagramas de clases.
- Construcción de repositorios base, con la arquitectura propuesta.
- Documentación de diseños
- Capacitación y entrega de diseños.

1.2 Descripción del problema

Actualmente, la división de arquitectura no cuenta con el personal suficiente para atender todos los proyectos en una ventana de tiempo apropiada, lo cual con frecuencia convierte a la etapa de diseño en un cuello de botella dentro del proceso global de desarrollo de sistemas. Esto incide negativamente en el tiempo de liberación de las soluciones, y a su vez, puede llevar a una disminución del nivel competitivo de la empresa.

Estos hechos, sumados a la necesidad constante de buscar el máximo nivel de automatización posible en todo proceso en el cual sea viable hacerlo, ha originado la necesidad de contar con una alternativa que permita obtener de forma oportuna un diseño inicial de los diagramas de componentes de software para una solución.

Este proyecto busca responder la siguiente pregunta de investigación:

¿Cuál es el grado de aceptación en la división de arquitectura, para una herramienta que genere automáticamente propuestas base de diagramas de componentes UML para una solución de software, a partir del ingreso de algunos requisitos?

1.3 Solución propuesta

El momento actual está marcado por el auge en el uso de la inteligencia artificial, concepto que fue acuñado hace ya más de medio siglo [1], pero que gracias a los avances en la tecnología del hardware [2], ha venido evolucionando en los últimos años, y actualmente nos brinda soporte en la realización de muchas de nuestras actividades diarias, no solo a nivel personal sino también a las empresas, las cuales en mayor o menor grado van descubriendo la necesidad de adaptar esta revolucionaria tecnología en los procesos que le generan más valor. Esta reciente revolución causada por la inteligencia artificial se debe en gran medida a la llegada a escena de los grandes modelos de lenguaje (LLM) [3]-[4], a los cuales podemos ver como complejos sistemas informáticos que son capaces de procesar y generar texto, basándose en datos de entrenamiento y generando respuestas tan coherentes que rivalizan con las respuestas proporcionadas por una persona [4]-[5].

La empresa objetivo de este proyecto, no solo que no escapa de esta tendencia, sino que sus propios objetivos empresariales le impulsan a estar siempre a la vanguardia en el uso de la tecnología y la aplicación de esta, en cada uno de sus procesos, en los cuales sea factible y viable hacerlo. Con base en lo expuesto, es evidente y mandatorio la necesidad de hacer uso de la inteligencia artificial con el fin de potenciar la eficiencia de los procedimientos de un departamento tan enfocado en la tecnología, como lo es el departamento de Software Factory y su división de arquitectura. Uno de estos procedimientos es el diseño de las

soluciones de software, y como un paso importante de este procedimiento se identifica la elaboración de los diagramas de componentes UML [6] para la propuesta de la solución de software.
La asistencia de la inteligencia artificial en la arquitectura de software es un tema que recientemente ha tomado relevancia y está siendo objeto de muchos estudios [7], los cuales en su mayoría se han enfocado en la generación de diagramas de clases [8]. Sin embargo, a la fecha de este trabajo no se encontró estudios enfocados en la generación de diagramas de componentes mediante el uso de los LLM y las técnicas de optimización existentes; RAG [9] y Fine-Tuning [10], siendo la creación de diagramas de componentes, un paso fundamental en el proceso de diseño de una solución de software y por ende muy importante para la división de arquitectura de la empresa objeto de estudio.

Con base en lo indicado anteriormente, y con la finalidad de lograr reducir los tiempos de entrega de las soluciones por parte de la división de arquitectura, se propone el diseño de una herramienta informática que permita generar diagramas base de componentes de software de forma ágil, mediante el ingreso de requisitos por parte de un usuario y haciendo uso de un gran modelo de lenguaje para la generación de dichos diagramas. Adicionalmente, para lograr resultados contextualmente relevantes, se propone aplicar alguna de las estrategias ya mencionadas. La finalidad de la propuesta es que un usuario técnico, ya sea un arquitecto de software o un líder de proyectos acceda a una herramienta web, en la cual pueda ingresar una serie de requisitos funcionales y/o no funcionales en lenguaje natural, para inmediatamente solicitar al sistema que genere un diagrama de componentes de software basado en dichos requisitos. Luego de la confirmación por parte del usuario, se esperaría que el sistema elabore la petición (considerando

los requisitos ingresados) en un formato entendible para un LLM local. Luego de esto se espera que el sistema envíe la petición hacia el LLM, el cual debería generar un diagrama de componentes de software bastante alineado a los requisitos y enmarcado en el contexto tecnológico de la empresa.
Es importante resaltar el hecho de usar un LLM local, ya que esto garantiza que la información que se compartirá con el LLM no saldrá de la empresa, asegurando de esta manera la protección de la información.

Para cumplir con esa finalidad se realizarán las actividades indicadas a continuación de forma general:

- Se realizará levantamiento de información con el personal de arquitectura, mediante el uso de un instrumento definido en el apartado de metodología. Este levantamiento nos permitirá determinar:
  - ✓ Cuáles son los criterios que se debe considerar en la definición de un adecuado contexto para la generación de diagramas de componentes de software.
  - ✓ Cuáles son las consideraciones que se deben tener respecto a la seguridad de la información.
  - ✓ Qué características o funcionalidades aportarían más valor a una herramienta generadora de diagramas de componentes de software.
  - ✓ Que formato de respuesta sería más adecuado como resultado de una petición a la herramienta.
- A continuación, se realizará el diseño de la herramienta, mediante la elaboración de diagramas de contexto, diagramas de componentes de software y diagramas de secuencia, basándonos un poco en la definición del framework C4 [11].

- Luego de contar con el diseño inicial, se realizará la elección de tecnologías para el prototipo, esto incluye FrontEnd, Backend, LLM, base de datos, etc.
- Finalmente se procederá con la elaboración de un prototipo funcional para poder realizar la presentación con los usuarios, con el objetivo de conseguir su aprobación para el diseño.

Luego del diseño de esta herramienta y su presentación al usuario, se espera toda la retroalimentación posible y una acogida favorable de parte de este, al conocer todo el valor que le puede aportar una herramienta de este tipo. Se espera que luego de este trabajo, se priorice un proyecto para la implementación de la herramienta propuesta.

1.4 Objetivo general

Diseñar una herramienta informática, que genere diagramas base de componentes de software de forma automática, mediante el ingreso de requisitos de usuario, haciendo uso de un gran modelo de lenguaje (local) para la elaboración del diagrama y aplicando técnicas para la mejora contextual de los resultados.

1.5 Objetivos específicos

- Realizar levantamiento de la situación inicial.
- Diseñar la arquitectura para la solución propuesta.
- Elaborar prototipo funcional.
- Realizar pruebas de generación de diagramas de componentes.
- Validar resultados con la división de arquitectura.

1.6 Metodología

Este proyecto será abordado con un enfoque cuantitativo, con un alcance descriptivo y un diseño no experimental transversal. Se adopta este diseño debido a que el proyecto no incluye el uso de experimentos. Lo que si se deberá realizar es la medición de la variable de interés al final del proyecto.

No se realizarán muestreos debido a que la población objeto de estudio tiene un universo de 6 individuos que cuentan con las siguientes características: Población que comprende a los colaboradores de la división de arquitectura, en el departamento de Software Factory, de una empresa de Telecomunicaciones que lleva sus operaciones en Guayaquil-Ecuador y se enfoca en el sector corporativo.

La variable de interés que se ha definido para el proyecto es el grado de aceptación que tiene el diseño de la herramienta propuesta, por parte del personal de la división de arquitectura del departamento de Software Factory.

Variable	Definición conceptual	Definición operacional
Grado de aceptación	La aceptación es la facultad por la cual una persona admite a otra persona, animal, objeto o pensamiento o la acción por la cual las recibe de manera voluntaria.	Se medirá luego de la presentación del diseño y se utilizará para su medición un instrumento de encuesta. La unidad de medida serán los puntos porcentuales.

Tabla 1: Variables de medición

Fuente: El autor

Para la recopilación de datos ya se tiene definidas las unidades de análisis, y como instrumento de medición se utilizará una encuesta con escala de Likert, lo cual brindará confiabilidad, validez y objetividad a la recopilación.

CAPÍTULO II

MARCO TEÓRICO

El objetivo de este capítulo es brindar un conjunto de definiciones, convenciones e información relevante que sirva como guía para la consecución exitosa del proyecto.

2.1 Lenguaje unificado de modelado (UML)

2.1.1 Introducción al lenguaje unificado de modelado

Un modelo es la representación de un objeto de la vida real, y dicho modelo es utilizado para entender y/o darle forma al objeto, explorar sus características y/o definirle nuevas características. En el caso de la ingeniería de software, estos modelos se gestionan mediante un lenguaje de modelado, como UML [6].

El lenguaje unificado de modelado o UML , como comúnmente se lo conoce, es un lenguaje de modelado estándar y visual, ampliamente utilizado y cuyo propósito es entre otros varios, permitir la especificación, diseño y documentación de sistemas de software, de forma clara y estandarizada [6]. Podemos considerar a UML como una interfaz o lenguaje ubicuo para la comunicación o entendimiento entre un diseñador y un desarrollador respecto a la estructura, relaciones y comportamiento de cada uno de los artefactos que componen una solución de software, ya que

mediante la aplicación de UML se pueden definir, modelar o plasmar todas estas características tan importantes de una solución informática.

Aunque los primeros esfuerzos por definir métodos para el desarrollo de software con lenguajes orientado a objetos se remontan a mediados de la década de 1980, no fue hasta 1997 que el Object Management Group (OMG), un consorcio creado en 1989 en Estados Unidos, con el objetivo de desarrollar estándares y especificaciones para la industria del software, presentó, luego de muchos meses de trabajo, la propuesta final de UML, gracias al trabajo en conjunto de expertos como Grady Booch, Ivar Jacobson y James Rumbaugh [6] y otros muchos metodólogos y expertos de software de aquella época. Es gracias a estos personajes, que hoy en día podemos contar con este estándar tan útil y tan ampliamente utilizado en toda la industria del desarrollo de software.

2.1.2 Objetivos del modelado UML

- Definir, entender y manipular la representación digital de objetos o conjuntos de objetos del mundo real que componen un sistema, capturando sus detalles clave, características, comportamientos y las relaciones entre cada uno de ellos.
- Ser un lenguaje abierto, de propósito general, no propietario, de tal forma que pueda ser utilizado por cualquier modelador.
- Ser un lenguaje universalmente aceptado por la gran comunidad del software, ya que hereda atributos de varios de los métodos de modelado más utilizados en la década de 1980 a 1990 como son los métodos OMT, Booch y Objectory, facilitando su adopción.

- Fomentar las buenas prácticas de diseño, como son la encapsulación y la separación de responsabilidades [6], entre otros.
- Ser un lenguaje altamente expresivo y universal para lograr manejar todos los conceptos posibles que surgen en los sistemas modernos, como por ejemplo la concurrencia [6].

2.1.3 Tipos de diagramas UML

En UML se definen varios tipos de vistas para un sistema, cada una de las cuales cuenta con su propia notación y se enfoca en un aspecto específico de dicho sistema. Pero en conjunto, estas vistas proporcionan una descripción integral del mismo [6].

A continuación, repasaremos cada una de dichas vistas y se dará una descripción muy literal de acuerdo con la literatura [6]:

2.1.3.1 Vista estática

Describe los aspectos estáticos o estructurales del sistema, como las clases, objetos y sus relaciones. Los diagramas definidos en esta clasificación son:

- **Diagrama de clases**

2.1.3.2 Vista de casos de uso

Representan el comportamiento funcional del sistema desde la perspectiva del usuario o actor. Los diagramas definidos en esta clasificación son:

- **Diagrama de casos de uso**

2.1.3.3 Vista de interacción

Describe cómo los objetos en el sistema interactúan entre sí a través del envío de mensajes. Los diagramas definidos en esta clasificación son:

- **Diagramas de secuencia**
- **Diagramas de interacción**

2.1.3.4 Vista de máquina de estados

Modela los estados por los que pasan los objetos del sistema y las transiciones entre esos estados como respuesta a eventos. Los diagramas definidos en esta clasificación son:

- **Diagrama de estados**

2.1.3.5 Vista de actividad

Describe los flujos de trabajo y procesos dentro del sistema, mostrando las actividades y las transiciones entre ellas. Los diagramas definidos en esta clasificación son:

- **Diagrama de actividades**

2.1.3.6 Vistas físicas

Se utilizan para modelar los aspectos físicos del sistema, tales como la configuración de hardware y software en la implementación. Los diagramas definidos en esta clasificación son:

- **Diagrama de componentes**
- **Diagrama de despliegue**

2.1.3.7 Vistas de gestión de modelos

Ayuda a gestionar la estructura organizativa del modelo, dividiendo el sistema en paquetes manejables. Los diagramas definidos en esta clasificación son:

- **Diagrama de paquetes**

A continuación, se abordará un poco más a detalle los diagramas de componentes, por tratarse de uno de los elementos clave del proyecto.

2.1.3.8 Diagramas de componentes

Los diagramas de componentes son un tipo de diagrama UML que describe cada uno de los elementos modulares que componen un sistema, y como estos componentes se relacionan entre sí. Cada componente del diagrama representa una unidad lógica que encapsula una funcionalidad particular del sistema. Los diagramas de componentes están formados por los siguientes elementos:

- **Componentes**

  Son elementos físicos y reemplazables del sistema, los cuales pueden ser módulos de código, bibliotecas, ejecutables, o bases de datos. Estos componentes generalmente se representan con una forma rectangular.

- **Paquetes**

  Se usan como contenedores de componentes.

- **Interfaces**

  Representan los puntos de comunicación que expone un componente para que pueda ser consumido por otro componente. Generalmente se representa como un círculo o como una línea con el nombre de la interfaz.

- **Puerto**

  Punto de interacción independiente entre el componente y su entorno.
- **Artefactos**

  Representan elementos físicos generados por el sistema, como por ejemplo archivos.
- **Relaciones de dependencia**

  Representan las relaciones entre los componentes del sistema, indicando si un componte utiliza a otro componente o depende de él. Las relaciones se definen con líneas de trazo.

Estos elementos de los diagramas de componentes se pueden apreciar en la siguiente imagen:

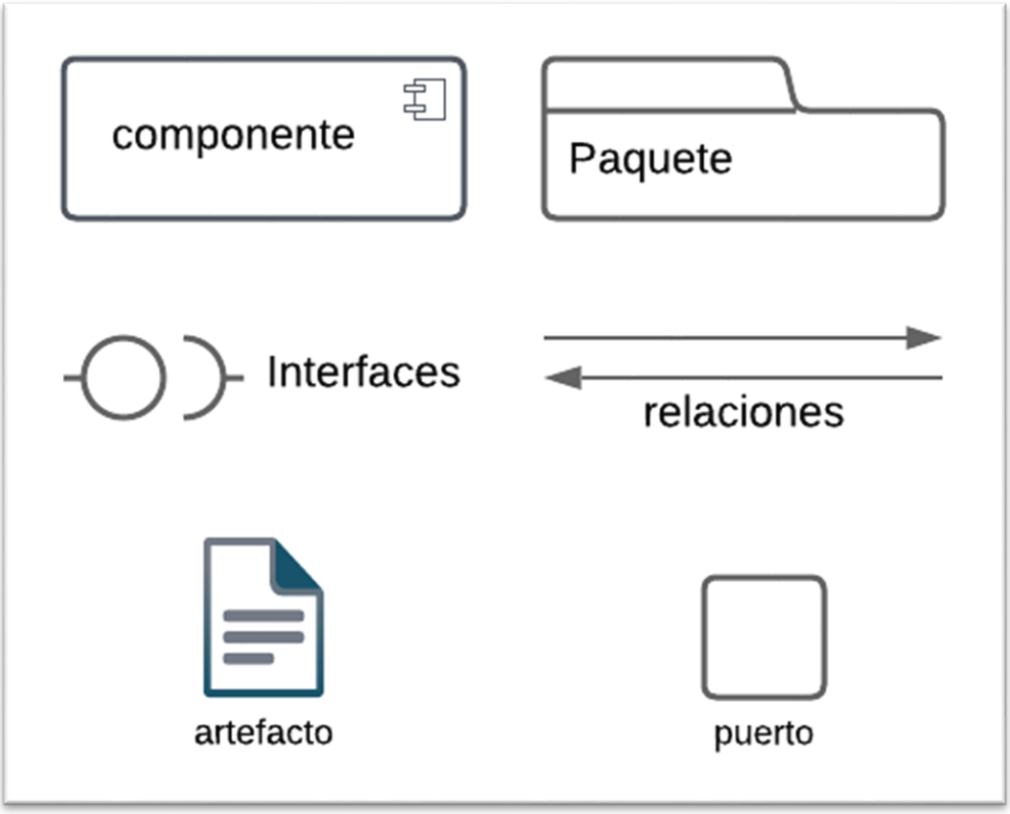


Figura II.1: Partes de un diagrama de componentes

Fuente: El autor

2.1.4 Aplicaciones y usos del lenguaje UML

El lenguaje de modelado unificado (UML) satisface un amplio espectro de necesidades en el contexto de la ingeniera y desarrollo de software, para conseguir una definición clara y precisa sobre la arquitectura, comportamiento y estructura de los sistemas, sobre todo cuando estos son complejos. Con base en lo expuesto en los numerales anteriores, algunas de las aplicaciones para UML son las siguientes [6]:

2.1.4.1 Diseño de software

El objetivo es poder representar de forma clara, la estructura interna de cada uno de los elementos que conforman un sistema, mediante el uso de componentes, módulos, clases e interfaces.

2.1.4.2 Gestión de requerimientos

El objetivo es, basado en los requerimientos levantados, poder representar los comportamientos o casos de uso de un sistema y como este interactúa con cada uno de sus principales actores, esto mediante el uso de diagramas de actividades y diagramas de casos de uso.

2.1.4.3 Modelado de sistemas

El objetivo es poder representar la estructura general de los sistemas y la interacción entre sus distintos elementos, para cada uno de los casos de uso definidos, esto mediante el uso de diagramas de componentes, diagramas de secuencia y diagramas de estados.

2.1.4.4 Diseño de bases de datos

El objetivo es poder representar las diferentes entidades de un modelo de datos para un sistema, incluyendo sus

atributos y la relación entre cada una de dichas entidades, además de sus cardinalidades y restricciones.

2.1.4.5 Documentación

El objetivo es hacer una recopilación de los diagramas implementados para el sistema y armar una documentación clara de la estructura general de la solución, así como la estructura interna de cada uno de sus elementos y la relación en interacción entre cada uno de ellos, de manera que cualquiera que conozca UML pueda leer esa documentación y entender el funcionamiento de dicho sistema.

2.1.5 Herramientas de modelado UML

En la actualidad, existe un sin número de herramientas que facilitan la elaboración de diagramas UML, cada una de ellas brinda una funcionalidad particular según su enfoque. A continuación, listamos algunas de estas herramientas:

2.1.5.1 Herramientas de generación gráfica

Son herramientas que permiten al usuario generar diagramas en formato gráfico, simplemente seleccionando y arrastrando los elementos que desean que formen parte del diagrama, entre ellas podemos nombrar a Visual Paradigm [12], Power Designer [13], etc.

2.1.5.2 Herramientas basadas en la nube

Son aquellas herramientas que permiten la generación de diagramas de forma remota y colaborativa, brindando al usuario la posibilidad de gestionar sus diagramas desde cualquier ubicación con acceso a internet,

algunas de estas herramientas son: Lucidchart [14], Gliffy [15], etc.

2.1.5.3 Herramientas con generación automática de código

Son herramientas que, además de permitir el modelamiento de diagramas, también permiten generar código a partir de dichos diagramas. Entre ellas podemos nombrar a StarUml [16], Modelio [17], etc.

2.1.5.4 Herramientas de modelado de procesos de negocio

Son herramientas enfocadas en el modelamiento de procesos de negocio. Entre ellas podemos nombrar a Camunda Modeler, Bizagi, Visual Paradigm [12], etc.

2.1.5.5 Herramientas de validación y simulación

Son herramientas que además de permitir el modelamiento de diagramas, también permiten simular y validar el comportamiento de estos modelos. En esa categoría podemos nombrar a USE (Uml-based Specification Environment) [18].

2.1.5.6 Herramientas de modelado basado en texto

Son herramientas que permiten generar modelos UML mediante descripciones textuales utilizando una sintaxis específica. El usuario describe los elementos del modelo y sus relaciones mediante la sintaxis de la herramienta y esta, a su vez en tiempo real, va renderizando esa sintaxis en una representación gráfica del modelo con formato png, svg u otro. Algunas de estas herramientas son: PlantUml [19], Mermaid [20], etc.

Un punto clave a resaltar en este tipo de herramientas es la facilidad que brindan para el versionamiento de los modelos, ya que al tratarse de texto es muy fácilmente

gestionable mediante una herramienta de control de versiones.

2.2 Arquitectura de software

2.2.1 Introducción a la arquitectura de software

La arquitectura de software es una disciplina fundamental en el desarrollo de sistemas informáticos complejos. Se enfoca en el conjunto de decisiones estructurales que definen y coordinan los componentes del sistema, así como las relaciones entre ellos. En esencia, la arquitectura de software define como un sistema se descompone en sus partes fundamentales, cómo estas partes se comunican entre sí, y de qué manera interactúan con el entorno [21].

Una arquitectura sólida es clave para asegurar la calidad de un sistema en términos de **mantenibilidad**, **escalabilidad**, **rendimiento** y **seguridad**, entre otros aspectos. Esta arquitectura debe ser diseñada teniendo en cuenta tanto los requisitos funcionales (lo que el sistema debe hacer) como los no funcionales (cómo el sistema debe comportarse), los cuales incluyen restricciones técnicas, decisiones de diseño y la calidad general del software [22].

Dentro del contexto del desarrollo de software, una arquitectura bien definida actúa como una hoja de ruta que guía a los equipos de desarrollo durante todo el ciclo de vida del sistema. Además, proporciona una base sólida para la toma de decisiones futuras, facilita la comunicación entre los involucrados y reduce los riesgos asociados con la evolución del sistema, como la deuda técnica y la complejidad innecesaria.

A medida que se desarrollaba la industria del software, seguramente entre muchos casos de error, pero muchos más de éxito, los diseñadores más expertos comenzaron a notar ciertas similitudes en problemáticas particulares que se presentaban entre un proyecto y otro. Los expertos se dieron cuenta que ciertas implementaciones podían abstraerse de los detalles y volverse genéricas, de manera que pudiesen ser aplicadas en muchos casos de uso similares [23]. A partir de ahí, se desarrollaron los conceptos de principios y patrones de diseño.

2.2.2 Principios de diseño

El diseño de software es una disciplina fundamental dentro del desarrollo de sistemas de información, pues determina cómo los componentes individuales se organizan e interactúan entre sí para formar un sistema integral, coherente y funcional. Para guiar este proceso, se han establecido una serie de principios de diseño que permiten la creación de arquitecturas robustas, escalables y mantenibles. Estos principios no solo facilitan el trabajo de los desarrolladores, sino que también garantizan que el sistema sea capaz de evolucionar sin perder su integridad [24]. Es común plasmar estos principios de diseño en las arquitecturas de software mediante las herramientas de modelado antes vistas. A continuación, nombramos varios de los principales principios de diseño más utilizados en nuestro entorno:

2.2.2.1 Principio de responsabilidad única

Sostiene que cada clase o módulo debe tener una única razón para cambiar, es decir, debe estar enfocado en una única tarea o responsabilidad. Este principio está estrechamente relacionado con la separación de responsabilidades y es crucial para la creación de

software mantenible y fácilmente comprensible. La adherencia a este principio permite evitar el acoplamiento innecesario entre módulos y promueve el desarrollo de componentes reutilizables [24].

2.2.2.2 Principio abierto/cerrado

Propuesto por Bertrand Meyer, establece que los módulos de software deben estar abiertos para la extensión, pero cerrados para la modificación. Esto significa que el comportamiento de un sistema puede ampliarse mediante la adición de nuevo código, sin alterar el código existente. El OCP es particularmente importante en sistemas que requieren mantenimiento continuo, ya que minimiza el riesgo de introducir errores en componentes ya funcionales cuando se incorporan nuevas características [24].

2.2.2.3 Principio de sustitución de Liskov

Establece que los objetos de una clase derivada deben poder reemplazar a los objetos de la clase base sin alterar el comportamiento del programa. Este principio asegura la correcta herencia entre clases, promoviendo la creación de jerarquías coherentes y funcionales. En términos prácticos, el LSP previene errores que pueden surgir cuando las subclases no cumplen con las expectativas establecidas por la clase base, garantizando así la integridad del sistema [24].

2.2.2.4 Principio de segregación de interfaces

Establece que ningún cliente debe estar obligado a depender de interfaces que no utiliza. En otras palabras, las interfaces grandes y generales deben dividirse en interfaces más pequeñas y específicas para que los

clientes solo necesiten conocer y utilizar los métodos que realmente les son útiles. Este principio ayuda a evitar la sobrecarga de responsabilidades en las interfaces y previene el denominado "code smell" que ocurre cuando un cambio en una interfaz afecta innecesariamente a muchas clases que dependen de ella. En la práctica, el ISP reduce la cantidad de dependencias innecesarias en un sistema, promoviendo un diseño más flexible y desacoplado. Cuando las interfaces son más específicas y cumplen con este principio, se facilita el mantenimiento y la evolución del código, ya que los cambios en una interfaz no afectarán a componentes que no tienen relación directa con ella [25].

2.2.2.5 Principio de inversión de dependencias

Establece que los módulos de alto nivel no deben depender de módulos de bajo nivel; ambos deben depender de abstracciones. Además, las abstracciones no deben depender de detalles concretos, sino que los detalles deben depender de las abstracciones. Este principio es clave para garantizar la flexibilidad y escalabilidad de un sistema, ya que permite desacoplar las implementaciones concretas de sus abstracciones, facilitando cambios en las capas bajas del sistema sin afectar a las capas superiores [25].

2.2.2.6 Principio Don’t Repeat yourself

Sugiere que cada pieza de conocimiento o lógica debe tener una representación única y no duplicada en el sistema. La repetición innecesaria de código aumenta la complejidad del mantenimiento y eleva el riesgo de

errores, ya que cualquier cambio en un componente duplicado debe realizarse en múltiples ubicaciones. Aplicar el DRY implica una organización eficiente del código y el uso adecuado de la abstracción para evitar duplicidades [24].

2.2.2.7 Principio KISS

Aboga por la simplicidad en el diseño, sugiriendo que los sistemas y componentes deben mantenerse tan simples como sea posible. La complejidad innecesaria suele llevar a sistemas difíciles de entender, mantener y escalar. Mantener el diseño sencillo no significa sacrificar la funcionalidad, sino asegurarse de que el sistema se desarrolla de la forma más clara y directa posible. Este principio es esencial para sistemas en crecimiento, donde la complejidad tiende a aumentar con el tiempo si no se controla adecuadamente [24].

Los principios de diseño constituyen los pilares fundamentales para crear software que sea adaptable, mantenible y flexible a lo largo del tiempo. Siguiendo estos principios, los desarrolladores logran una estructura clara y bien organizada, facilitando el crecimiento del sistema sin comprometer su estabilidad. Sin embargo, a medida que los sistemas crecen en complejidad, los principios de diseño pueden no ser suficientes para abordar ciertos problemas recurrentes de forma específica. Aquí es donde los **patrones de diseño** desempeñan un papel crucial.

2.2.3 Patrones de diseño

En el desarrollo de software, los patrones de diseño son soluciones repetibles a problemas comunes que surgen durante la fase de diseño y arquitectura de componentes. Estos patrones

encapsulan las mejores prácticas y lecciones aprendidas por expertos a lo largo del tiempo, ofreciendo a los diseñadores y desarrolladores una guía estructurada para enfrentar desafíos específicos sin reinventar la rueda en cada proyecto [23]. Los patrones de diseño no son implementaciones concretas, sino esquemas que pueden ser adaptados y personalizados según el contexto de cada sistema. Su objetivo principal es mejorar la eficiencia del diseño, la reutilización de soluciones y la comunicación entre los equipos de desarrollo al proporcionar un lenguaje común.

El uso de los patrones de diseño es especialmente valioso cuando se trata de abordar problemas de diseño recurrentes, como la administración de dependencias, la flexibilidad en la creación de objetos o la gestión de interacciones complejas entre componentes. Los patrones de diseño normalmente se plasman en una arquitectura de software en los diagramas de clase, mediante alguna de las herramientas de modelado ya revisadas.

Aunque los patrones de diseño proporcionan soluciones valiosas, es fundamental entender que no son recetas universales. La correcta aplicación de un patrón depende del contexto y las necesidades específicas del sistema en desarrollo [26]. Un uso inapropiado o forzado de patrones puede introducir complejidad innecesaria y generar más problemas que beneficios. Por lo tanto, su implementación debe ser cuidadosa y reflexiva, alineándose con los principios de diseño que rigen la arquitectura del sistema.

Cada patrón de diseño responde a un tipo específico de problema y se clasifica según la naturaleza de la solución que ofrece. A continuación, se da una breve descripción de los patrones que son más utilizados en nuestro día a día, agrupados según su clasificación. Se deja fuera de este listado a aquellos patrones

para los cuales, las herramientas de desarrollo actuales ya brindan implementaciones rápidas, como por ejemplo los patrones builder, singleton, etc.:

2.2.3.1 Patrones creacionales

Tratan sobre la forma en que los objetos son creados y gestionados. A continuación, nombramos algunos de ellos:

2.2.3.1.1 Factory Method

Este patrón define una interfaz para crear objetos en una clase base, pero permite que las subclases alteren el tipo de objeto que se creará. En lugar de instanciar directamente los objetos, el patrón delega la responsabilidad de la creación a las subclases, promoviendo el principio de responsabilidad única y la separación de la lógica de creación de la lógica de negocio. Este enfoque permite que las clases base no estén acopladas a clases específicas de productos, facilitando la adición de nuevas variantes de productos sin cambiar el código que utiliza esas clases [23].

2.2.3.1.2 Abstract Factory

Este patrón va un paso más allá del Factory Method y se utiliza cuando es necesario crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. A través de una interfaz común, el patrón permite que un conjunto de objetos (o productos) se creen de manera conjunta,

asegurando que todos pertenezcan a la misma familia o contexto de creación. El objetivo es proporcionar una solución que permita crear varios objetos que deben ser compatibles entre sí, sin que el código que los utiliza tenga conocimiento de sus clases concretas [23].

2.2.3.2 Patrones estructurales

Se centran en la composición y relación entre objetos y clases. A continuación, se nombra algunos de ellos:

2.2.3.2.1 Adapter

Este patrón actúa como un intermediario que permite que clases con interfaces incompatibles trabajen juntas. Su principal función es traducir la interfaz de una clase existente para que otra clase pueda utilizarla sin necesidad de modificar su código. Es comúnmente utilizado cuando se quiere integrar una nueva clase en un sistema existente sin alterar la estructura de este último. El adaptador convierte las llamadas de métodos esperadas por el cliente en las llamadas equivalentes a los métodos de la clase que necesita ser adaptada, garantizando que ambas partes puedan interactuar de manera armoniosa [23].

2.2.3.2.2 Bridge

Este patrón separa la abstracción de la implementación, permitiendo que ambas evolucionen de manera independiente. En

lugar de acoplar fuertemente la abstracción a su implementación, el patrón Bridge utiliza una composición para mantener la independencia entre estas dos capas. Esto es útil cuando se espera que un sistema crezca con múltiples variantes de abstracción e implementación, ya que cada una puede modificarse sin afectar la otra [23].

2.2.3.2.3 Decorator

Este patrón permite añadir responsabilidades adicionales a un objeto de manera dinámica, sin alterar su estructura original. A diferencia de la herencia, que añade funcionalidad en tiempo de diseño, el Decorador lo hace en tiempo de ejecución. Cada decorador "envuelve" al objeto original y extiende su comportamiento, proporcionando una alternativa flexible a la subclasificación para añadir funcionalidades incrementales [23].

2.2.3.2.4 Facade

Este patrón proporciona una interfaz simplificada y unificada para un conjunto de interfaces en un subsistema complejo. El objetivo de la fachada es ocultar la complejidad del subsistema, proporcionando un punto de acceso único que es fácil de usar y entender. Aunque el sistema subyacente puede tener muchas funcionalidades, el patrón Facade expone solo las más

relevantes para el usuario final, facilitando su uso [23].

2.2.3.2.5 Proxy

Este patrón actúa como un sustituto o intermediario de otro objeto, controlando el acceso a este. El proxy puede proporcionar funcionalidad adicional, como control de acceso, carga diferida o gestión de recursos, sin modificar el objeto original. Existen varios tipos de proxy, como el proxy remoto, que representa a un objeto ubicado en otra máquina, o el proxy virtual, que retrasa la creación de un objeto pesado hasta que realmente sea necesario [23].

2.2.3.3 Patrones de comportamiento

Abordan la interacción y responsabilidad entre los objetos. Estas categorías permiten organizar y aplicar los patrones de manera sistemática, ayudando a los arquitectos de software a identificar rápidamente el enfoque adecuado para un problema concreto. A continuación, se nombra algunos de ellos:

2.2.3.3.1 Strategy

Este patrón permite definir una familia de algoritmos, encapsular cada uno de ellos y hacer que sean intercambiables. A través de este patrón, el comportamiento de un objeto puede variar según el algoritmo que se utilice, sin modificar su estructura. El patrón Strategy se aplica cuando existen varias formas de realizar una tarea y el cliente necesita

seleccionar entre ellas de manera dinámica [23]. Este patrón normalmente trabaja de la mano con algún patrón de factoría para obtener la estrategia a utilizar.

2.2.3.3.2 Template Method

Este patrón define la estructura básica de un algoritmo en una clase abstracta, permitiendo que las subclases implementen o modifiquen ciertos pasos específicos del proceso sin cambiar la estructura general del algoritmo. Este patrón permite reutilizar el código común para todos los algoritmos, dejando que las subclases personalicen las partes que varían [23].

2.2.3.3.3 Mediator

Este patrón centraliza la comunicación entre los objetos en un sistema, evitando que los objetos se comuniquen directamente entre sí. En lugar de tener interacciones complejas y dependencias entre varios componentes, cada uno de ellos se comunica a través de un mediador. Esto reduce el acoplamiento entre las clases y simplifica el mantenimiento y la evolución del sistema. El mediador conoce las interacciones que deben ocurrir entre los objetos y coordina esas interacciones de manera eficiente [23].

2.2.3.3.4 Chain of Responsability

Este patrón permite pasar una solicitud a través de una cadena de manejadores potenciales hasta que uno de ellos se haga

cargo de procesarla. Cada manejador en la cadena decide si puede procesar la solicitud o si la pasa al siguiente manejador. Este patrón desacopla al remitente de la solicitud de su receptor, permitiendo que más de un objeto tenga la oportunidad de procesarla. También facilita la adición o modificación de los manejadores sin afectar a los demás [23].

2.2.4 Patrones de arquitectura

Los patrones de arquitectura son soluciones generales y repetibles para problemas comunes que surgen al diseñar **la estructura global** de un sistema de software. A diferencia de los patrones de diseño, que abordan problemas más específicos a nivel de clase o componente, los patrones de arquitectura proporcionan un marco para la organización de sistemas completos. Su objetivo es garantizar que el software sea escalable, mantenible y flexible frente a los cambios y que pueda cumplir con los requisitos técnicos y de negocio de manera eficiente.

Estos patrones establecen una guía clara sobre cómo deben organizarse y relacionarse los diferentes componentes de un sistema, facilitando la toma de decisiones arquitectónicas que conduzcan al éxito del proyecto. A continuación, se describen algunos de los patrones de arquitectura más utilizados y relevantes en el desarrollo de sistemas de software modernos:

2.2.4.1 Arquitectura en capas

Es uno de los patrones más tradicionales y utilizados. En este patrón, el sistema se divide en capas jerárquicas, donde cada una de ellas tiene una responsabilidad

claramente definida. Las capas más comunes incluyen la capa de presentación (interfaz de usuario), la capa de negocio (lógica de la aplicación), la capa de acceso a datos y la capa de almacenamiento de datos. Cada capa interactúa únicamente con la capa inmediatamente inferior, lo que permite una mayor modularidad y separación de responsabilidades [27].

2.2.4.2 Arquitectura hexagonal

Este patrón de arquitectura, propuesto por Alistair Cockburn, busca resolver algunos de los problemas de acoplamiento que existen en patrones más tradicionales, como la arquitectura en capas. En lugar de depender fuertemente de las capas internas y externas, la arquitectura hexagonal organiza el sistema en torno a una unidad central (el dominio de la aplicación) rodeada de puertos y adaptadores. Los puertos son interfaces que definen cómo interactuar con la aplicación, mientras que los adaptadores son implementaciones concretas que se conectan a estos puertos. Este enfoque desacopla completamente el núcleo del negocio de las capas externas, como la base de datos o las interfaces de usuario, lo que hace que el sistema sea más flexible y adaptable.

Uno de los beneficios clave de la arquitectura hexagonal es que facilita las pruebas y la evolución del sistema. Dado que las dependencias de la aplicación (como la infraestructura o las interfaces de usuario) están aisladas a través de adaptadores, es mucho más sencillo reemplazar o modificar partes del sistema sin afectar al núcleo de la aplicación [28].

2.2.4.3 Arquitectura de tuberías y filtros

En este tipo de arquitectura, el procesamiento de datos se descompone en una secuencia de pasos independientes (filtros), cada uno de los cuales realiza una transformación sobre los datos. Los filtros están conectados mediante canales (tuberías) que transportan los datos de un paso a otro. Este patrón es especialmente útil en sistemas que requieren un procesamiento de datos continuo, como en los flujos de datos en tiempo real o los sistemas de compilación [29].

2.2.4.4 Arquitectura orientada a eventos

Este es un patrón de arquitectura en el que los componentes del sistema se comunican entre sí mediante la emisión y recepción de eventos. En lugar de estar directamente acoplados, los componentes reaccionan a eventos emitidos por otros sin depender de su estado o ejecución. Este enfoque es útil para sistemas que requieren un alto grado de desacoplamiento y reactividad, como aplicaciones de tiempo real, comercio electrónico o sistemas de monitoreo [30].

2.2.4.5 Arquitectura de microservicios

Este patrón es un enfoque moderno que promueve la construcción de aplicaciones a partir de servicios pequeños y autónomos, cada uno centrado en una funcionalidad específica del negocio. Estos servicios están desacoplados y pueden desarrollarse, implementarse y escalarse de manera independiente. Cada microservicio se comunica con los demás a través de interfaces bien definidas, como Apis REST o

mensajería, lo que permite una mayor flexibilidad y escalabilidad horizontal. Como pueden observar este patrón hace uso de varias de las bondades de los anteriores patrones [31].

2.2.4.6 Arquitectura monolítica modular

Esta arquitectura es un enfoque en el que un sistema se desarrolla como una única aplicación, pero internamente está organizado en módulos independientes que encapsulan diferentes funcionalidades. A diferencia del monolito tradicional, donde todos los componentes están estrechamente acoplados y cualquier cambio en una parte del sistema puede afectar a otras, en el monolito modular los módulos están bien delimitados y se comunican entre sí a través de interfaces claras y desacopladas, como si fueran pequeños servicios dentro de un mismo monolito.

Este enfoque permite organizar el sistema de forma modular sin la complejidad inherente a los microservicios, donde cada módulo es independiente pero **no necesita** desplegarse ni gestionarse como un servicio separado. Pero al mismo tiempo su estructura brinda la posibilidad de migrar con relativa facilidad, cada módulo a un microservicio si fuese necesario [32]-[33].

2.2.5 Diseño y documentación de la arquitectura

El diseño y la documentación de la arquitectura de software son aspectos fundamentales en la construcción de sistemas complejos. Un buen diseño arquitectónico garantiza que el sistema sea escalable, mantenible y adaptable a los cambios

futuros, mientras que la documentación proporciona una guía clara para los desarrolladores y otros stakeholders sobre cómo está estructurado el sistema, sus componentes y cómo interactúan entre sí [21].

Uno de los principales desafíos en la documentación arquitectónica es encontrar un equilibrio entre la complejidad y la claridad. Si la documentación es demasiado abstracta, podría resultar poco útil para los desarrolladores; por otro lado, si es demasiado detallada, puede ser difícil de mantener y navegar. En este contexto, el modelo C4 se ha convertido en una herramienta poderosa y ampliamente adoptada para abordar estos problemas, proporcionando un enfoque estructurado y claro para la documentación de la arquitectura.

2.2.5.1 Modelo C4

El modelo C4, creado por Simon Brown, es una técnica para describir la arquitectura de software utilizando diferentes niveles de abstracción. Su nombre hace referencia a las cuatro capas o vistas que utiliza para representar el sistema de forma progresivamente más detallada: Contexto, Contenedores, Componentes y Código (Clases). El propósito principal de C4 es proporcionar un enfoque escalonado que facilite la comprensión tanto del panorama general del sistema como de los detalles técnicos específicos cuando sea necesario [34]. El modelo C4 está constituido por 4 vistas principales, las cuales se detallan a continuación:

- **Diagrama de contexto**

  Es la vista más alta de la arquitectura. Muestra el sistema en su conjunto y cómo interactúa con su

entorno, como otros sistemas o actores externos (usuarios, clientes, etc.). Esta vista es útil para comunicar a los stakeholders no técnicos, la funcionalidad del sistema y cómo se conecta con otras partes del ecosistema tecnológico.

Por ejemplo, en una aplicación web, el diagrama de contexto podría mostrar cómo interactúan los usuarios con el sistema, los sistemas de autenticación de terceros y los servicios de datos externos [11].

- **Diagrama de contenedores**

  Descompone el sistema en diferentes contenedores de software, como aplicaciones web, bases de datos, servicios Backend, etc. Este nivel permite ver los componentes principales del sistema y cómo interactúan entre sí a través de protocolos o Apis.

  Esta vista es particularmente útil para arquitectos y desarrolladores que necesitan entender la estructura del sistema y cómo se distribuyen las responsabilidades entre diferentes contenedores de software [11].

- **Diagrama de componentes**

  Se adentra en un nivel más profundo, mostrando los componentes individuales dentro de cada contenedor. Los componentes representan unidades funcionales más pequeñas dentro del sistema, como módulos o servicios que cumplen con una tarea específica dentro del contenedor [11].

  Este nivel es crucial para los equipos de desarrollo, ya que describe cómo están organizadas las

funcionalidades internas y cómo los componentes interactúan entre sí dentro de un contenedor.

- **Diagrama de código (o clases)**
  Es el más detallado y muestra la implementación de un componente específico en términos de clases o estructuras de código. Aunque no siempre es necesario documentar a este nivel, puede ser útil para describir las interacciones dentro de componentes críticos o altamente complejos [11].
  Este nivel suele ser más dinámico, ya que el código cambia con más frecuencia que el diseño de alto nivel. Es importante mantener esta vista sincronizada con el estado actual del código base si se decide utilizarla.

2.2.5.2 Ventajas del Modelo C4

El modelo C4 presenta varias ventajas que lo hacen particularmente útil en el diseño y documentación de arquitecturas [34]:

- **Claridad y Simplicidad**
  Cada nivel del modelo C4 está diseñado para ser lo suficientemente claro como para que cualquier stakeholder pueda entender el sistema desde su respectiva perspectiva. Los niveles de abstracción aseguran que tanto los desarrolladores como los stakeholders no técnicos tengan la información que necesitan.
- **Flexibilidad y Escalabilidad**
  C4 permite comenzar desde una vista general y luego profundizar en detalles técnicos específicos

cuando sea necesario. Esto facilita la escalabilidad del diseño, ya que los arquitectos pueden añadir nuevos componentes o contenedores sin perder de vista el panorama general del sistema.

- **Documentación Evolutiva**
  A medida que el sistema evoluciona, el modelo C4 facilita la actualización de la documentación sin necesidad de rediseñar todo el sistema. Los cambios se pueden reflejar solo en el nivel correspondiente, lo que hace que la documentación sea más fácil de mantener.
- **Comunicación Efectiva**
  Al proporcionar diferentes vistas de la arquitectura, el modelo C4 facilita la comunicación entre equipos multifuncionales. Los ejecutivos pueden comprender el contexto general del sistema, mientras que los desarrolladores tienen acceso a detalles más profundos sobre los componentes y el código.

2.2.5.3 Consideraciones en la Aplicación del Modelo C4

Si bien el modelo C4 proporciona una estructura clara para la documentación de la arquitectura, su aplicación efectiva requiere ciertas consideraciones [34]:

- **Mantener la Documentación Actualizada**
  Dado que la documentación arquitectónica tiende a volverse obsoleta si no se actualiza regularmente, es importante integrar la actualización de los diagramas C4 en el ciclo de vida del proyecto, especialmente cuando hay cambios importantes en el diseño o la implementación.

- **Adaptación al Contexto**

  El modelo C4 es flexible, pero es importante adaptar el nivel de detalle según el tamaño y la complejidad del sistema. En sistemas pequeños, puede que no sea necesario llegar hasta el nivel de clases, mientras que, en sistemas grandes y complejos, este nivel puede ser crucial.

2.3 Prototipos de software

2.3.1 Introducción a los prototipos de software

El prototipado de software es una técnica que permite el desarrollo de modelos preliminares o versiones simplificadas de un sistema antes de su implementación completa. Estos prototipos ayudan a los equipos de desarrollo a obtener una visión clara y comprensible del producto final, al tiempo que permiten la interacción con los usuarios desde etapas tempranas. Mediante la creación de versiones que reflejan características clave o flujos de trabajo esenciales, los prototipos facilitan la validación de requisitos y la identificación de mejoras o ajustes necesarios, antes de incurrir en los costos y esfuerzos asociados a la implementación total del sistema [35].

El uso del prototipado en software ha ganado relevancia en metodologías ágiles y en entornos donde la adaptabilidad y la retroalimentación continua son esenciales para el éxito del proyecto. Los prototipos no solo ayudan a los desarrolladores y diseñadores a visualizar la funcionalidad propuesta, sino que también permiten a los usuarios y stakeholders participar activamente en la validación y refinamiento de las funcionalidades y la experiencia de usuario [36].

2.3.2 Importancia de prototipado en un proyecto de software

El prototipado es una fase crucial en el ciclo de vida del desarrollo de software, ya que proporciona una forma tangible de representar una solución antes de que se realice la implementación completa. Este enfoque permite que los desarrolladores, clientes y usuarios finales interactúen con una versión preliminar del producto, lo que facilita la validación temprana de requisitos y expectativas. De esta manera, los prototipos actúan como un puente entre la conceptualización abstracta y la ejecución práctica del sistema, ofreciendo una visión más clara de cómo funcionará el software una vez finalizado [35].

Uno de los principales beneficios del prototipado es que reduce el riesgo de malentendidos o interpretaciones erróneas entre los diferentes actores involucrados en el proyecto. Al ofrecer una representación visual y, en algunos casos, interactiva de las funcionalidades principales, los usuarios pueden identificar problemas o discrepancias antes de que el software esté completamente desarrollado. Esto contribuye a que se realicen ajustes y mejoras en las primeras fases, lo que disminuye el costo de rectificar errores en etapas avanzadas del desarrollo [35].

Además, el prototipado permite una retroalimentación continua por parte de los usuarios y clientes, lo que es especialmente valioso en entornos donde los requisitos pueden cambiar o evolucionar rápidamente.

El valor del prototipado también radica en su capacidad para facilitar una mejor planificación del proyecto. Al identificar las posibles dificultades técnicas, de diseño o de experiencia de usuario a través de la creación de prototipos, los equipos de

desarrollo pueden anticiparse a problemas y ajustar la planificación del proyecto en consecuencia. Esto reduce los riesgos y mejora la estimación de tiempos y recursos [35].

Por último, el prototipado promueve la creatividad y la innovación. Dado que los equipos tienen la libertad de explorar diferentes enfoques y soluciones sin el compromiso inmediato de implementar un producto final, pueden probar ideas nuevas y realizar cambios iterativos sin consecuencias significativas en el presupuesto o en el cronograma [37].

2.3.3 Tipos de prototipos de software

En el desarrollo de software, existen diferentes tipos de prototipos que pueden ser utilizados en función de los objetivos del proyecto, el presupuesto y el nivel de detalle necesario. Los prototipos de software se clasifican en varias categorías, que van desde representaciones básicas hasta simulaciones funcionales avanzadas, cada una con sus propios beneficios y aplicaciones. A continuación, detallamos tres de los tipos de prototipo más utilizados en nuestro contexto:

2.3.3.1 Prototipos de Baja Fidelidad:

Los prototipos de baja fidelidad son representaciones simplificadas y rápidas del sistema, a menudo en forma de bocetos o wireframes. Estos prototipos no suelen incluir funcionalidad interactiva ni detalles de diseño, pero permiten una visualización inicial de la estructura y el flujo de trabajo del software. Son muy útiles para obtener retroalimentación temprana de los usuarios sobre la interfaz y la disposición de los elementos, sin invertir grandes recursos en su creación [37].

Herramientas como Balsamiq y Figma permiten crear rápidamente estos prototipos.

2.3.3.2 Prototipos de Alta Fidelidad:

A diferencia de los de baja fidelidad, los prototipos de alta fidelidad se asemejan más al producto final tanto en funcionalidad como en diseño. Estos modelos pueden incluir interacciones, navegabilidad y elementos gráficos detallados, lo que facilita una comprensión más precisa del comportamiento del sistema. Se utilizan principalmente cuando es necesario validar aspectos específicos como la experiencia de usuario o el diseño visual [37]. Herramientas como Adobe XD, InVision, y Axure son frecuentemente empleadas para este tipo de prototipos.

2.3.3.3 Prototipos Funcionales:

En este tipo de prototipo, se desarrollan versiones que no solo incluyen el diseño, sino también partes del código funcional del software. Estos prototipos permiten a los equipos de desarrollo y a los usuarios probar características clave del sistema en un entorno realista, incluyendo interacciones, conectividad y desempeño. Son particularmente útiles cuando es necesario validar requisitos técnicos o de rendimiento antes de pasar a una implementación completa [38].

Cada uno de estos tipos de prototipos tiene un propósito específico en el ciclo de desarrollo del software, y la elección del tipo adecuado depende de las necesidades del proyecto, el grado

de incertidumbre en los requisitos, y el nivel de detalle requerido por los stakeholders.

2.3.4 Herramientas y tecnologías de desarrollo

En el proceso de desarrollo de software, las herramientas utilizadas son fundamentales para facilitar el trabajo de los equipos, mejorar la productividad y garantizar la calidad del producto final. Estas herramientas abarcan una amplia gama de funciones, desde la gestión de versiones del código hasta el diseño de interfaces y la automatización de pruebas. A continuación, se describen las herramientas seleccionadas para la elaboración del prototipo funcional de este proyecto:

2.3.4.1 Java

Es un lenguaje de programación versátil y orientado a objetos, desarrollado por Sun Microsystems en 1995 y actualmente mantenido por Oracle. Su principal ventaja es el concepto de "escribir una vez, ejecutar en cualquier lugar", lo que permite que el código escrito en Java funcione en cualquier plataforma que tenga una Máquina Virtual de Java (JVM). Su manejo automático de memoria y estructura modular lo hacen adecuado para una amplia variedad de aplicaciones, desde sistemas empresariales y servidores Backend hasta aplicaciones móviles [39].

Java también cuenta con un extenso ecosistema de bibliotecas y frameworks, como Spring e Hibernate, que simplifican el desarrollo de aplicaciones robustas y escalables. Su popularidad se debe no solo a su

rendimiento en entornos empresariales, sino también a su gran comunidad de desarrolladores y la abundancia de recursos literarios.

2.3.4.2 Springboot

Es un marco de trabajo basado en Spring que facilita el desarrollo de aplicaciones Java mediante la simplificación de la configuración y la infraestructura. Diseñado para crear aplicaciones listas para producción de manera rápida, Spring Boot elimina la necesidad de configuraciones manuales complejas al ofrecer configuraciones automáticas y una amplia gama de dependencias preconfiguradas. Esto permite a los desarrolladores centrarse en la lógica de negocio en lugar de en la configuración del entorno. Además, Spring Boot incluye un servidor embebido (como Tomcat), lo que permite ejecutar aplicaciones directamente sin necesidad de configuraciones adicionales [40].

Gracias a su capacidad de integración con otros proyectos de Spring, como Spring Data, Spring Security, y Spring Modulith, SpringBoot es ideal para crear aplicaciones empresariales escalables, microservicios y hasta monolitos modulares. Su popularidad radica en su enfoque pragmático y su extensa documentación oficial [40].

2.3.4.3 Intellij Idea Comunity Edition

Es una versión gratuita y de código abierto del entorno de desarrollo integrado (IDE) IntelliJ IDEA, desarrollado por JetBrains. Está diseñado para facilitar el desarrollo

de aplicaciones en varios lenguajes, especialmente Java y Kotlin, y proporciona un conjunto completo de herramientas que incluyen soporte para sistemas de control de versiones como Git, integración con Maven y Gradle, y funcionalidades de depuración avanzadas. Aunque no incluye características empresariales avanzadas disponibles en la versión Ultimate, la Community Edition es una excelente opción para desarrolladores que trabajan en proyectos de código abierto o desarrollo de aplicaciones estándar. Este IDE destaca por su interfaz intuitiva y sus potentes capacidades de autocompletado y refactorización de código, que aceleran significativamente el flujo de trabajo de los programadores [41].

2.3.4.4 Angular

Es un marco de trabajo de código abierto desarrollado por Google, diseñado para la creación de aplicaciones web de una sola página (SPA, por sus siglas en inglés) de alta calidad. Está construido sobre **TypeScript**, un superconjunto de JavaScript que añade tipado estático y otras características avanzadas, lo que mejora la escalabilidad y el mantenimiento de las aplicaciones. Gracias a TypeScript, los desarrolladores pueden detectar errores en tiempo de compilación, lo que mejora la calidad del código y reduce problemas en etapas posteriores.

Angular permite a los desarrolladores construir interfaces de usuario dinámicas y ricas en funcionalidades al proporcionar herramientas para la

vinculación de datos bidireccional, inyección de dependencias y gestión eficiente de componentes. Una de sus características más destacadas es su enfoque en el desarrollo modular, lo que permite dividir las aplicaciones en piezas reutilizables, facilitando su mantenimiento y escalabilidad.

Este framework es ampliamente utilizado por su capacidad para integrar funciones avanzadas como enrutamiento, formularios reactivos y manejo de estados. Además, Angular cuenta con una comunidad activa y una extensa documentación oficial que facilita el aprendizaje y la implementación [42].

2.3.4.5 Visual Studio Code

Es un editor de código fuente, gratuito y de código abierto, desarrollado por Microsoft. Popular por su ligereza y versatilidad, está diseñado para soportar una amplia variedad de lenguajes de programación como JavaScript, Python, Java, C++, y muchos más. VS Code se destaca por su entorno altamente personalizable, gracias a su vasta biblioteca de extensiones que permiten agregar características adicionales como depuración, control de versiones con Git, y soporte para frameworks específicos como Node.js o React.

Su interfaz amigable y la integración de herramientas avanzadas, como IntelliSense (autocompletado inteligente de código) y el terminal integrado, hacen que sea una excelente opción para los desarrolladores. Además, su comunidad activa y la abundancia de recursos lo hacen accesible para todos [43].

2.3.4.6 Python

Es un lenguaje de programación conocido por su simplicidad y legibilidad, lo que lo convierte en una opción ideal tanto para principiantes como para desarrolladores experimentados. Creado por Guido van Rossum en 1991, Python se caracteriza por una sintaxis clara y fácil de aprender, que permite a los programadores enfocarse más en resolver problemas que en la complejidad del lenguaje en sí. Gracias a su versatilidad, Python es utilizado en una amplia gama de aplicaciones, desde desarrollo web hasta análisis de datos, inteligencia artificial y automatización de tareas.

Una de las mayores fortalezas de Python es su comunidad activa y el extenso ecosistema de bibliotecas y herramientas disponibles. Esto facilita el desarrollo rápido de soluciones en campos como la ciencia de datos o el aprendizaje automático [44].

2.3.4.7 PyCharm Community Edition

Es una versión gratuita y de código abierto del entorno de desarrollo integrado (IDE) PyCharm, creado por JetBrains. Esta edición está diseñada especialmente para programadores que trabajan con Python y ofrece muchas de las características esenciales que hacen que PyCharm sea tan popular, como la edición inteligente de código, autocompletado, depuración y soporte para pruebas unitarias. Aunque carece de algunas de las herramientas avanzadas de la versión profesional, sigue

siendo una opción potente para aquellos que buscan un IDE confiable para proyectos en Python.

Lo que hace a PyCharm Community Edition tan atractivo es su facilidad de uso y su integración con herramientas como Git para el control de versiones, así como con entornos virtuales para gestionar dependencias. Para la elaboración de un prototipo esta edición gratuita cubre la mayoría de las necesidades del desarrollo Python. La documentación oficial y una comunidad activa hacen que sea fácil encontrar soluciones a problemas comunes [45].

2.3.4.8 Milvus DB

Es una base de datos de código abierto diseñada específicamente para la gestión y búsqueda de datos vectoriales, lo que la convierte en una herramienta clave en proyectos de inteligencia artificial y machine learning. Desarrollada por Zilliz, Milvus se destaca por su capacidad para manejar grandes volúmenes de datos no estructurados, como imágenes, videos y textos convertidos en representaciones vectoriales. Esta base de datos está optimizada para búsquedas rápidas y eficientes en datos de alta dimensionalidad, lo que la hace ideal para aplicaciones como el reconocimiento facial, la búsqueda de similitudes o el análisis de datos complejos.

Uno de los puntos fuertes de Milvus es su escalabilidad y su capacidad para integrarse con otras herramientas y frameworks de aprendizaje automático. Esto permite a los desarrolladores crear aplicaciones que involucren

búsqueda vectorial y procesamiento de grandes cantidades de datos sin preocuparse por el rendimiento. Milvus cuenta con una comunidad activa y una documentación detallada que facilita su implementación [46].

2.4 Grandes modelos de lenguaje

2.4.1 Introducción a los grandes modelos de lenguaje

Los Grandes Modelos de Lenguaje (LLMs) han revolucionado el campo de la inteligencia artificial al permitir la comprensión y generación de texto de manera natural y fluida. Estos modelos, como el GPT desarrollado por OpenAI, se basan en arquitecturas de redes neuronales profundas y se entrenan con enormes volúmenes de datos textuales [47]. Esto les permite captar patrones lingüísticos complejos, lo que los hace útiles en tareas como la traducción, la redacción de contenido y la generación de código. Uno de los avances clave que ha hecho posible el desarrollo de LLMs es la arquitectura de transformadores, presentada por Vaswani en 2017, que optimiza el procesamiento del lenguaje natural al manejar dependencias contextuales en secuencias largas de texto [48].

Los transformers son una arquitectura de redes neuronales que emplea un mecanismo de atención, lo que les permite enfocarse en diferentes partes de una oración o texto según la necesidad contextual. Este mecanismo de "autoatención" es fundamental para que los LLMs comprendan relaciones complejas entre palabras, frases y oraciones completas, lo que incrementa la capacidad del modelo para generar respuestas coherentes y contextualmente relevantes [48]. Antes de los transformers, las

arquitecturas de redes neuronales tradicionales presentaban limitaciones para procesar grandes secuencias de texto de manera eficiente. Gracias a los transformers, los modelos como GPT-3 y GPT-4 pueden procesar cantidades masivas de datos en paralelo, acelerando el aprendizaje y mejorando la calidad de las predicciones [49].

El entrenamiento de los LLMs sigue un enfoque de preentrenamiento y ajuste fino. Inicialmente, los modelos son expuestos a grandes corpus de datos no supervisados, donde aprenden patrones generales del lenguaje. Luego, se ajustan con tareas específicas para mejorar su rendimiento en actividades concretas, como la clasificación de texto o la respuesta a preguntas. Esta combinación de preentrenamiento y ajuste fino convierte a los LLMs en herramientas versátiles en áreas como la automatización de tareas, los asistentes virtuales y el análisis avanzado de datos, permitiendo aplicaciones que antes requerían mucha intervención humana [47].

Los LLMs, impulsados por la arquitectura de transformers, están siendo adoptados en muchos sectores, incluyendo salud, educación y software, y su impacto sigue en expansión.

2.4.2 Arquitectura de los grandes modelos de lenguaje

La arquitectura de los Grandes Modelos de Lenguaje (LLMs) se basa principalmente en los transformers, una innovación clave que ha permitido a estos modelos manejar grandes cantidades de texto y producir resultados coherentes. Los transformers utilizan un mecanismo llamado "autoatención", que permite al modelo enfocarse en distintas partes de una secuencia de texto según el contexto. A diferencia de los modelos previos, como las redes neuronales recurrentes (RNN) o convolucionales (CNN), los

transformers pueden procesar grandes secuencias de datos de forma paralela, lo que aumenta su eficiencia y rendimiento, especialmente cuando se trabaja con cantidades masivas de datos [48].

El mecanismo de autoatención es el corazón de los transformers. Este proceso permite que el modelo asigne un peso diferente a cada palabra de la secuencia, dependiendo de su relevancia con respecto a otras palabras. Así, el modelo no solo "recuerda" las palabras recientes, sino que también puede relacionar términos que están distantes en una oración o un párrafo [48]. Esta capacidad de capturar dependencias a largo plazo es esencial para generar texto que sea coherente y relevante en contextos complejos, lo que le da a los LLMs una ventaja significativa en tareas como la traducción automática, la generación de resúmenes y la creación de respuestas a preguntas.

En la práctica, la arquitectura de los transformers se compone de capas de codificadores y decodificadores, donde el codificador procesa la entrada inicial y el decodificador genera la salida [48]. Estos modelos se entrenan a través de técnicas como el preentrenamiento, que los expone a grandes corpus de texto sin necesidad de etiquetas, seguido del ajuste fino para tareas específicas. Durante este proceso, los transformers aprenden a predecir la siguiente palabra en una secuencia basándose en el contexto, lo que les permite adaptarse a una variedad de aplicaciones del lenguaje natural.

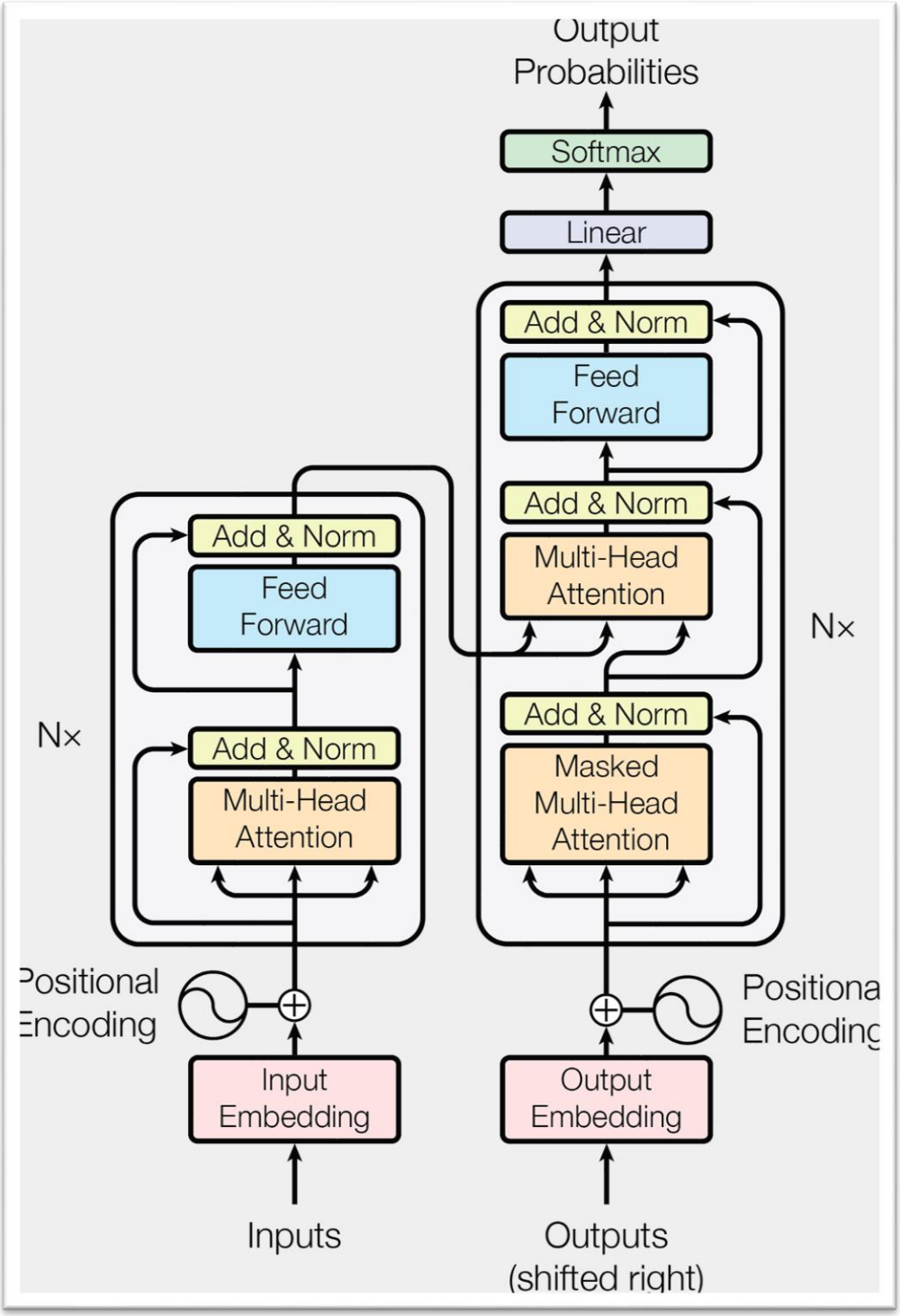


Figura II.2: Arquitectura Transformer

Fuente: [48]

En la figura 2.2, se puede apreciar la arquitectura Transformer propuesta por Vaswani en su publicación “Attention is all you need” en 2017.

2.4.3 Valor de los grandes modelos de lenguaje

El valor de los Grandes Modelos de Lenguaje (LLMs) radica en su capacidad para transformar la manera en que interactuamos con la información, facilitando procesos complejos que antes requerían una considerable intervención humana. Uno de sus principales aportes es la habilidad de generar y comprender lenguaje natural de manera fluida, lo que permite automatizar tareas como la redacción de textos, la traducción entre idiomas y la generación de resúmenes. Esto ha permitido a las organizaciones y empresas aumentar su eficiencia al reducir el tiempo y los recursos necesarios para producir contenido y procesar grandes volúmenes de información.

Los LLMs también aportan un valor significativo en áreas como la atención al cliente, la creación de contenido personalizado y la investigación científica. En servicios de atención al cliente, los modelos como GPT pueden proporcionar respuestas inmediatas y precisas a las consultas de los usuarios, mejorando la experiencia del cliente y liberando recursos humanos para tareas más complejas. En el ámbito científico y académico, su capacidad para analizar y generar información a partir de grandes corpus de datos facilita el descubrimiento de nuevas tendencias y conexiones que podrían pasar desapercibidas. Esto está impactando disciplinas como la biología, la medicina y las ciencias sociales, donde se utilizan para la extracción de información clave y la generación de hipótesis basadas en datos.

El valor de estos modelos no solo reside en sus aplicaciones actuales, sino también en su potencial para seguir mejorando a medida que se entrenan con más datos y se ajustan a nuevas necesidades. A medida que la tecnología avanza, se espera que los LLMs jueguen un papel aún más importante en áreas como la educación personalizada, la generación automatizada de código y la toma de decisiones empresariales. Además, plataformas como Hugging Face [50], han democratizado el acceso a estos modelos, permitiendo a desarrolladores y empresas de cualquier tamaño aprovechar sus capacidades sin la necesidad de infraestructuras complejas.

2.4.4 Aplicación de los grandes modelos de lenguaje en el diseño de software

Los Grandes Modelos de Lenguaje (LLMs) están teniendo un impacto cada vez mayor en el diseño y desarrollo de software, brindando herramientas potentes para automatizar y optimizar diversos aspectos del proceso de desarrollo. Una de las aplicaciones más notables es la generación automática de código, donde los LLMs pueden interpretar descripciones textuales y convertirlas en fragmentos de código funcional. Modelos como Codex de OpenAI, integrado en plataformas como GitHub Copilot, permiten a los desarrolladores escribir menos código manualmente y concentrarse en la lógica del negocio, ya que el modelo puede sugerir o incluso completar funciones completas basadas en la descripción de lo que se quiere lograr [51]. Esto ha mejorado tanto la productividad como la precisión en la creación de software.

Otra aplicación clave está en la documentación automática y mantenimiento de sistemas. Los LLMs pueden generar o

actualizar la documentación técnica de un sistema basándose en el código existente, haciendo que el proceso sea menos tedioso para los desarrolladores [52]. Además, en la fase de mantenimiento, los LLMs pueden ayudar a identificar errores en el código o sugerir mejoras basadas en patrones aprendidos durante su entrenamiento. Esto no solo reduce el tiempo que los equipos dedican a depurar software, sino que también mejora la calidad del producto final al hacer recomendaciones basadas en buenas prácticas y soluciones probadas [53].

En cuanto al diseño arquitectónico de software, los LLMs también están empezando a desempeñar un papel significativo. Algunas aplicaciones que utilizan estos modelos, pueden analizar código existente y generar diagramas de arquitectura de software automáticamente [54], lo que es útil para documentar aplicaciones complejas o sistemas heredados. Además, pueden ayudar a los arquitectos de software a explorar diferentes opciones de diseño, sugiriendo alternativas basadas en el análisis de grandes volúmenes de datos de software previamente diseñados. Herramientas como PlantUml, cuando se integran con LLMs, permiten a los desarrolladores y arquitectos visualizar componentes clave y sus relaciones de manera automática a partir de descripciones textuales o código fuente.

2.4.5 Aumentar la relevancia contextual de los resultados

En algunas ocasiones, los resultados que proporciona un LLM no son tan exactos como se esperaría, debido a la generalidad de los datos con los cuales fue entrenado. En estos escenarios suele ser muy útil aplicar ciertas técnicas con las cuales se puede lograr una mejora sustancial en la calidad contextual de los resultados del LLM. Para conseguir esta mejora en la relevancia contextual en

los resultados generados por un Gran Modelo de Lenguaje (LLM), es fundamental optimizar el proceso de recuperación y filtrado de información. A continuación, se indican dos técnicas comúnmente empleadas para el mejoramiento de los resultados de un LLM:

2.4.5.1 Generación Aumentada por Recuperación (RAG)

En esta técnica, el LLM se combina con una base de datos estructurada que proporciona información adicional basada en las consultas del usuario. Este enfoque permite al modelo trabajar con datos precisos y específicos en lugar de depender únicamente de su preentrenamiento, mejorando la calidad de las respuestas generadas [55]. Por ejemplo, el LLM puede acceder a documentos relevantes o a bases de datos internas en tiempo real, enriqueciendo así la respuesta con contexto específico.

- **Ventajas**:
  - Actualización dinámica: Permite acceder a información actualizada en tiempo real.
  - Mayor precisión: Combina respuestas generativas con información exacta de bases de datos.
  - Reducción de alucinaciones: Minimiza respuestas incorrectas al tener fuentes de datos directas.
- **Desventajas**:
  - Dependencia de fuentes: Requiere bases de datos bien estructuradas.

- Costos computacionales: El acceso y procesamiento de datos externos puede impactar en el rendimiento.

2.4.5.2 Ajuste fino del modelo (Fine-Tuning)

Al re-entrenar el modelo con ejemplos que reflejen los contextos y temas más comunes dentro de un área particular, como el diseño de software y el ecosistema tecnológico propio de una organización, el LLM puede generar respuestas más alineadas con las necesidades del usuario [56]. Además, el uso de memorias de contexto a largo plazo permite que el modelo retenga y reutilice información relevante de interacciones anteriores, mejorando la coherencia en conversaciones prolongadas.

- **Ventajas**:
  - Adaptación específica: Ajusta el modelo a dominios particulares, mejorando la relevancia en áreas específicas.
  - Mejora la coherencia: Genera respuestas más alineadas con el campo de aplicación.
- **Desventajas**:
  - Requiere datos etiquetados: Necesita conjuntos de datos bien preparados y etiquetados.
  - Actualización limitada: El modelo no se adapta en tiempo real a nueva información.
  - La creación de nuevos modelos cada vez que realiza un ajuste fino implica un alto consumo de recursos.

En estudios recientes [10]-[57]-[58] sobre fine-tuning y RAG en los Grandes Modelos de Lenguaje (LLMs), se han encontrado resultados mixtos. Aunque el fine-tuning puede mejorar el rendimiento en ciertos dominios, su eficacia no siempre es consistente, especialmente cuando se utiliza en combinación con RAG con pocas muestras de datos (menos de 1000). En lugar de mejorar la calidad de las respuestas, el fine-tuning puede tener un impacto negativo en estos casos. Esto ocurre porque la cantidad limitada de datos no es suficiente para entrenar el modelo adecuadamente, reduciendo la precisión de las respuestas generadas. Las evaluaciones humanas muestran que no existe una técnica universalmente superior, y la efectividad depende tanto del modelo base como de la precisión de los datos externos utilizados.

2.4.6 Limitaciones y desafíos de los LLM en el diseño de software

A pesar de los avances logrados por los Grandes Modelos de Lenguaje (LLMs) en el diseño de software, existen limitaciones y desafíos importantes. Una de las principales barreras es la falta de comprensión profunda del contexto. Los LLMs, aunque buenos para generar fragmentos de código, no siempre captan la arquitectura completa o la lógica subyacente de un sistema, lo que puede resultar en soluciones parciales o incorrectas. Además, estos modelos son propensos a generar respuestas erróneas o "alucinaciones" cuando los datos de entrenamiento no son suficientes o adecuados para la tarea.

Otra limitación significativa es que los LLMs dependen de datos de entrenamiento estáticos, lo que significa que pueden quedarse desactualizados rápidamente en entornos de desarrollo de software que evolucionan constantemente. Sin la capacidad de aprender en tiempo real, los LLMs requieren reentrenamientos periódicos, lo que puede ser costoso tanto en términos de tiempo como de recursos computacionales. Además, el fine-tuning, aunque útil en ciertos escenarios, no garantiza mejoras en todas las situaciones, y su implementación conlleva un alto costo inicial y una alta dependencia de conjuntos de datos etiquetados con precisión.

En cuanto a los desafíos operacionales, los modelos a gran escala requieren una infraestructura robusta para ser desplegados y gestionados, lo cual es especialmente problemático cuando se integran con técnicas como RAG, que implican el uso constante de bases de datos externas. Esta integración puede afectar la eficiencia del sistema y complicar el proceso de generación de respuestas precisas.

A partir de los estudios recientes [8], se ha comprobado que aunque los LLMs pueden generar diagramas, como los de clases UML de manera comparable a los diagramas creados manualmente, presentan limitaciones en la precisión semántica. Esta dificultad para captar relaciones semánticas complejas resalta la necesidad de avanzar en el desarrollo de tecnologías de IA más sofisticadas que puedan ofrecer una comprensión más profunda de los vínculos semánticos. Los LLMs pueden ser útiles para el prototipado rápido de diagramas, pero requieren mejoras para llegar a automatizar completamente el proceso de diseño arquitectónico.

2.5 Trabajos similares

2.5.1 Revisión de trabajos similares

- Uno de los trabajos relevantes en la automatización de la generación de diagramas UML es el propuesto por Abdelkareem M. Alashqar, titulado "Automatic Generation of UML Diagrams from Scenario-based User Requirements" [59]. Este estudio aborda el uso de procesamiento de lenguaje natural (NLP) para la generación automática de diagramas UML, en particular diagramas de clases y de secuencia, a partir de requisitos de usuario escritos en lenguaje natural. Alashqar propone un algoritmo que utiliza técnicas de NLP para identificar actores, objetos y las interacciones entre ellos, permitiendo así la creación automática de estos diagramas a partir de escenarios de usuario. El estudio concluye que el sistema desarrollado, llamado AGUML, es capaz de mejorar la eficiencia en las fases de análisis y diseño de sistemas, reduciendo el tiempo necesario para la documentación visual. Sin embargo, el autor también señala mejoras, como la posibilidad de permitir el ingreso de texto más estructurado para obtener resultados óptimos y la necesidad de considerar otros tipos de diagramas.
- Otro trabajo reciente en este campo es la tesis de Daniele De Bari, titulada "Evaluating Large Language Models in Software Design: A Comparative Analysis of UML Class Diagram Generation" [8]. En este estudio, se evaluó la capacidad de los LLMs para generar diagramas de clases UML, comparándolos con los creados manualmente por las personas. Se concluye que los LLMs generan diagramas precisos en lo sintáctico y pragmático, pero tienen dificultades en la precisión semántica.

Desde una perspectiva práctica, los LLMs son útiles para el diseño iterativo y prototipado rápido, aunque todavía se necesitan avances para automatizar completamente el proceso. Se propone investigar más sobre cómo integrar LLMs en herramientas de desarrollo de software para mejorar la comprensión de enlaces semánticos y la automatización del diseño de software .

- Otro trabajo reciente, relacionado al campo de la generación de diagramas UML mediante el uso de LLM es "From Image to UML: First Results of Image-Based UML Diagram Generation using LLMs" de Aaron Conrardy y Jordi Cabot [60]. Este estudio explora el uso de LLMs multimodales, como GPT-4V, para convertir imágenes de diagramas de clases UML en modelos formales utilizando la notación PlantUml. Aunque los resultados son prometedores, el estudio destaca la necesidad de intervención humana debido a errores sintácticos y semánticos en los resultados generados.
- Otro trabajo importante en este ámbito es "Generating UML Class Diagram from Natural Language Requirements: A Survey of Approaches and Techniques" [61]. Este estudio proporciona un análisis exhaustivo de los enfoques y técnicas utilizados para la generación automática de diagramas de clases UML a partir de requisitos escritos en lenguaje natural. Se exploran diferentes metodologías, como el procesamiento de lenguaje natural y las técnicas de recuperación de información, para convertir los requisitos en estructuras formales. El estudio destaca los avances en la automatización del diseño de software, pero también subraya las limitaciones actuales en la comprensión semántica completa y la necesidad de

intervención humana para corregir ambigüedades y errores en los diagramas generados.

- El siguiente trabajo relacionado es "How LLMs Aid in UML Modeling: An Exploratory Study with Novice Analysts" [62]. Este estudio examina cómo los modelos de lenguaje grande, como GPT-3, pueden ayudar a los analistas novatos en la creación de modelos UML, como diagramas de casos de uso, diagramas de clases y diagramas de secuencia. Los resultados sugieren que, si bien los LLMs son útiles para generar estos diagramas, existen limitaciones, las cuales no son del todo claras en dicho artículo.
- El último trabajo relacionado que se analizó es "Large Language Models for Software Engineering: A Systematic Literature Review" de Xinyi Hou y sus colaboradores [7]. Este estudio realiza una revisión sistemática del uso de los LLMs en tareas de ingeniería de software, identificando sus aplicaciones más efectivas, como la generación de código y la documentación. A través de 229 estudios revisados entre 2017 y 2023, el trabajo destaca los principales desafíos y oportunidades que los LLMs presentan en este campo, incluyendo la optimización del procesamiento de datos y la evaluación de rendimiento.

2.5.2 Identificación de vacíos en el conocimiento

A partir de la revisión de trabajos similares, se identifican varios vacíos en el conocimiento. Uno de los principales es la falta de precisión semántica en la generación automática de diagramas UML, donde los LLMs aún no capturan con total exactitud las relaciones complejas entre entidades. Además, la integración de

técnicas multimodales, como la conversión de imágenes a diagramas UML, todavía requiere intervención humana significativa. Otra área insuficientemente explorada es el impacto a largo plazo de las mejoras en LLMs mediante fine-tuning y su capacidad para adaptarse a dominios específicos sin incurrir en altos costos computacionales.

Otro vacío adicional identificado es la falta de estudios que exploren la generación automática de diagramas de componentes utilizando LLMs. Aunque los trabajos actuales abordan mayormente la creación de diagramas de clases, la automatización de diagramas de componentes no ha recibido suficiente atención. Dada la naturaleza textual de las respuestas generadas por LLMs, herramientas como PlantUml ofrecen una oportunidad para transformar esas descripciones en diagramas visuales, pero este enfoque aún no ha sido plenamente abordado en la literatura actual. La falta de investigaciones sobre esta integración abre espacio para futuras investigaciones que optimicen este proceso.

2.5.3 Conclusión de revisión de trabajos similares

La revisión de trabajos similares destaca importantes avances en la aplicación de LLMs para la generación de diagramas UML y tareas de ingeniería de software, pero también revela limitaciones clave. Aunque los modelos actuales pueden automatizar partes del proceso de diseño, aún enfrentan desafíos en cuanto a precisión semántica y manejo de datos complejos. La necesidad de intervención humana y la dependencia de grandes conjuntos de datos para el fine-tuning son obstáculos que requieren más investigación. Estas limitaciones abren la puerta a futuras

investigaciones que busquen expandir y mejorar la eficiencia y precisión de los LLMs en este campo.

CAPÍTULO III

DEFINICIÓN DE LA SITUACIÓN ACTUAL

El objetivo de este capítulo es determinar la situación actual del proceso de diseño de diagramas de componentes de software, mediante el modelado AS-IS, la descripción de sus componentes, herramientas utilizadas y limitaciones identificadas, así como la definición de criterios de aceptación y alcance de este.

3.1 Descripción del proceso actual de diseño de arquitectura de software

3.1.1 Flujo para el diseño de diagramas de componentes

El flujo para el diseño del diagrama de componentes de software se desarrolla en varias etapas clave:

3.1.1.1 Levantamiento

Inicialmente, se lleva a cabo, en conjunto con el equipo de desarrollo, analistas de sistemas y stakeholders, un levantamiento exhaustivo de los requisitos, en donde se identifican las necesidades de los usuarios, el alcance y visión del proyecto, así como restricciones operativas que pudiesen ser definidas.

3.1.1.2 Análisis

A continuación, se realiza un análisis detallado, en el que se examinan cada uno de los requisitos levantados, con la finalidad de:

- Validar y equilibrar las necesidades del negocio frente a la factibilidad técnica y operacional.
- Identificar componentes clave que formarán parte de la solución. Estos componentes pueden ser componentes ya existentes en el ecosistema tecnológico de la empresa o compontes nuevos que deban ser creados para la solución.
- Especificar cómo los diferentes componentes del sistema interactúan entre sí, definiendo claramente las interfaces, dependencias y flujos de datos necesarios.
- Identificar posibles desafíos técnicos que podrían surgir en la implementación y definir estrategias para reducir su impacto. Esto incluye definir si es necesario realizar alguna prueba de concepto debido a la implementación de alguna nueva tecnología.
- Asegurarse de que la solución propuesta no solo cumpla con los requisitos actuales, sino que también sea capaz de manejar un crecimiento futuro en términos de usuarios, datos o funcionalidades.
- Asegurarse que la solución cumpla al menos con los estándares mínimos de seguridad establecidos por el departamento de seguridad lógica.
- Tratar de detectar posibles amenazas que podrían afectar el éxito del proyecto, como problemas

técnicos, cambios en los requisitos, limitaciones de recursos o problemas de integración.

- Asegurar la resiliencia del sistema, tratando de que el diseño contemple posibles fallos y esté preparado para manejar eventualidades, mejorando así la estabilidad y continuidad del sistema.

3.1.1.3 Diseño

Posteriormente, se procede a la elaboración y documentación del diagrama, que proyecta gráficamente la estructura interna del sistema, mediante un conjunto de componentes y sus conexiones.

3.1.1.4 Socialización

Finalmente, el resultado es socializado con los equipos de desarrollo, calidad, DBA y operaciones, en una sesión de trabajo que permite a todos los equipos compartir sus dudas u observaciones de mejora para la arquitectura propuesta.

3.1.2 Modelo AS-IS

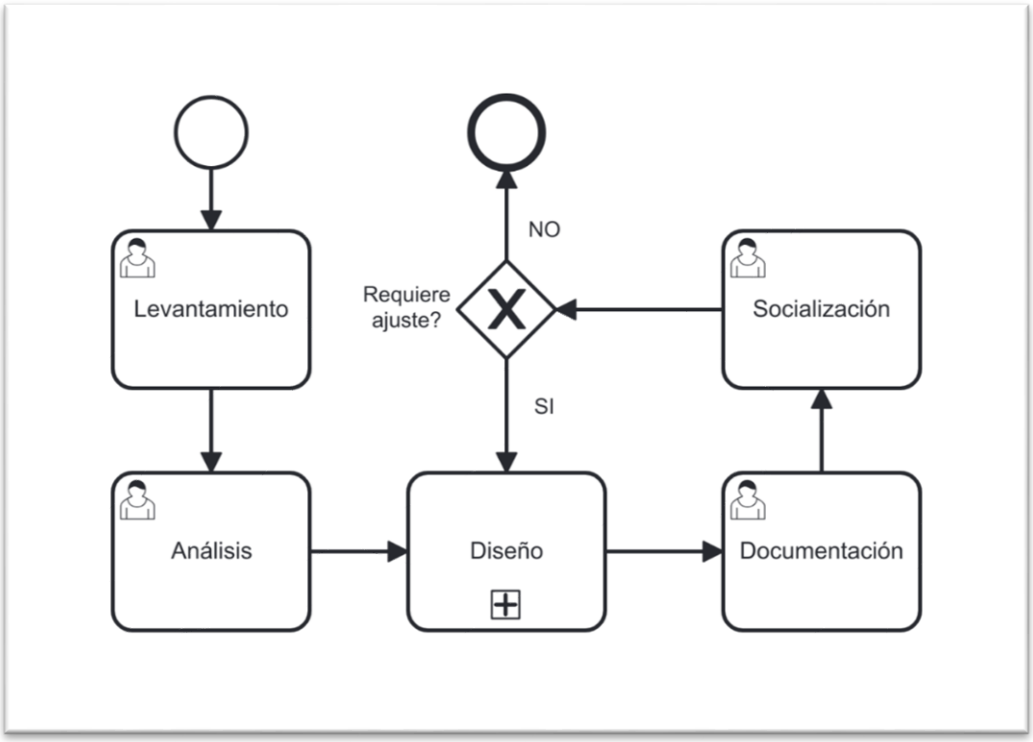


Figura III.1: Modelo AS-IS general del flujo de diseño

Fuente: El autor

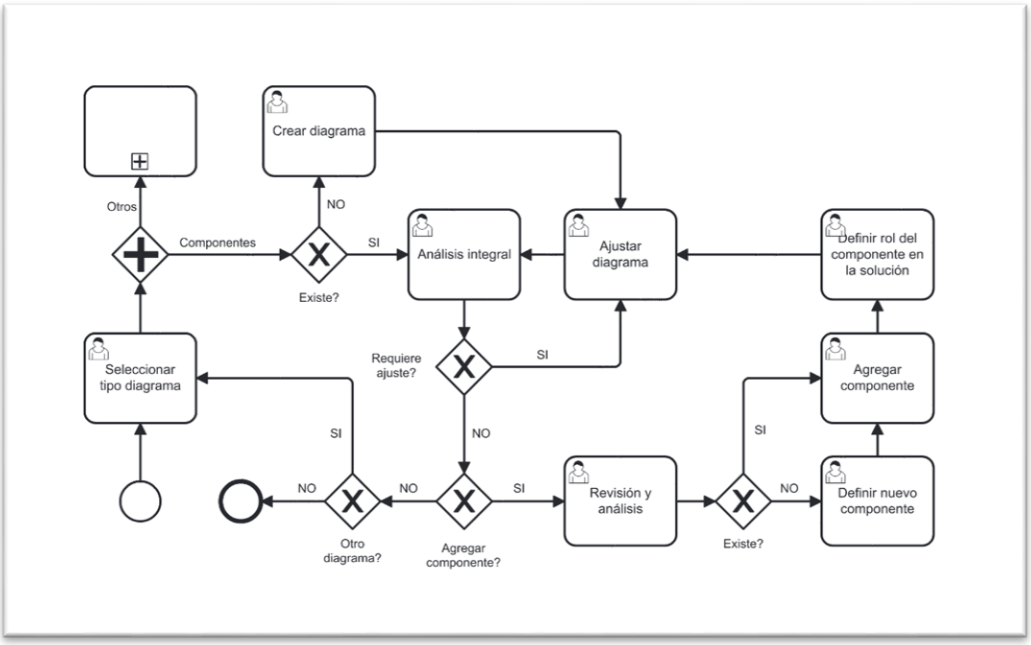


Figura III.2: Detalle de la actividad “Diseño” del diagrama anterior

Fuente: El autor

En las imágenes 3.1 y 3.2, se puede apreciar de forma visual, el procedimiento o flujo que se detalló en la sección anterior.

3.1.3 Herramientas y técnicas

Lucidchart es le herramienta que actualmente se utiliza para la elaboración de los diagramas de arquitectura. Como ya se indicó en el respectivo capítulo, Lucid es una plataforma ampliamente adoptada para la creación de diagramas visuales, ya que permite una representación clara y organizada de los diferentes componentes de software, proporcionando un entorno colaborativo que facilita la actualización y revisión en tiempo real por parte de los equipos de desarrollo y arquitectura.

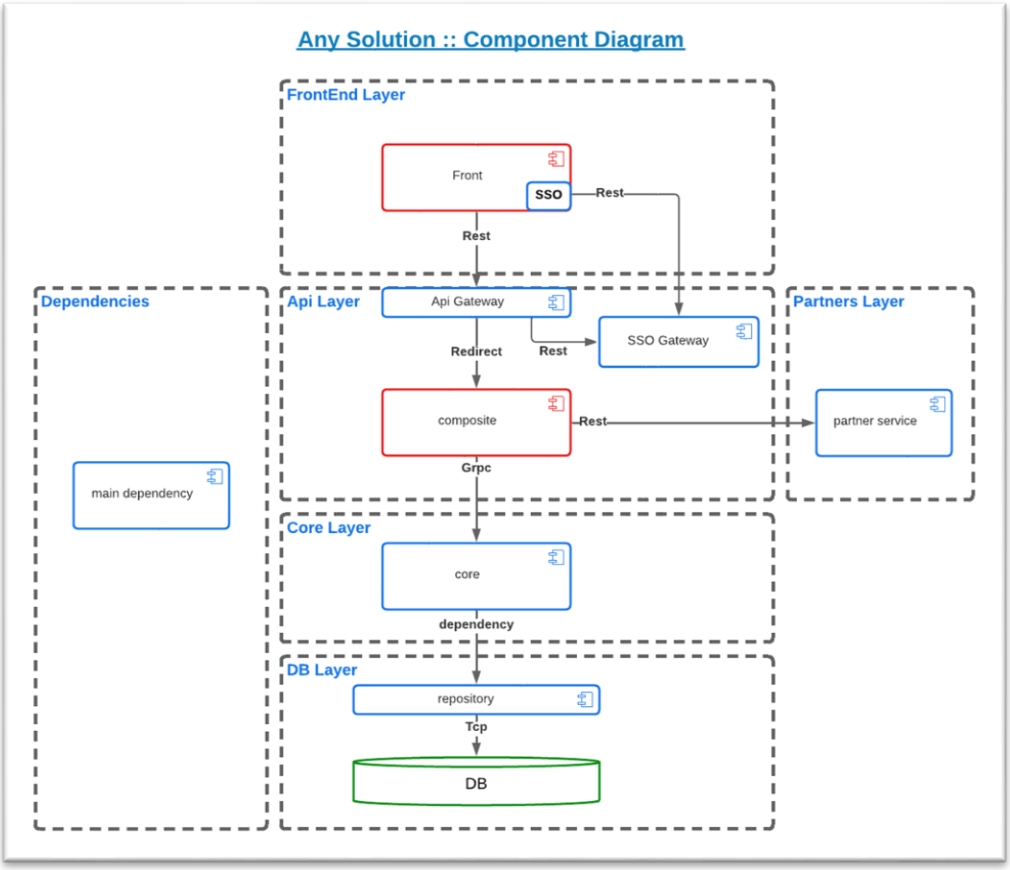


Figura III.3: Diagrama creado con Lucidchart

Fuente: El autor

En cuanto a las técnicas empleadas en el diseño, como se puede apreciar en la figura 3.3, se sigue un enfoque modular y estructurado por capas, donde cada componente está agrupado según su funcionalidad dentro de una capa específica, como la Capa “FrontEnd”, “API”, “Core”, entre otras, lo que permite una separación lógica y una mayor claridad en la visualización de responsabilidades y dependencias. El uso de relaciones bien definidas entre los módulos, como “Rest”, “Grpc” y “Tcp”, ayuda a

ilustrar cómo los componentes interactúan entre sí y cuáles son los protocolos utilizados para su comunicación.

Además, se aplica principios como la descomposición jerárquica de los componentes, donde se identifican dependencias claras y flujos de datos bien establecidos, minimizando la complejidad. Este enfoque permite que el diagrama refleje tanto la arquitectura técnica como el flujo funcional de la solución. La implementación de módulos comunes, como generadores de diseño y utilidades compartidas, también sigue un enfoque de reutilización de componentes, asegurando la escalabilidad y mantenibilidad de la arquitectura a largo plazo.

3.1.4 Roles involucrados

El proceso de diseño de diagramas de componentes de software involucra diferentes fases en las que varios actores desempeñan roles clave. Estos roles establecen quiénes son responsables, quiénes deben ser consultados o informados, y quiénes tienen la autoridad final en cada etapa, lo cual resulta muy útil con miras a definir una matriz RACI, cuya terminología se describe en la siguiente tabla:

Término	Descripción
R	Letra asignada a la persona que realiza la actividad.
A	Letra asignada a la persona con la responsabilidad final sobre la tarea.
C	Letra asignada a la persona que debe ser consultada.
I	Letra asignada a la persona que solo deber ser informada.

Tabla 2: Terminología RACI

Fuente: El autor

Una vez que se ha definido la terminología de una matriz RACI, se describen los roles involucrados en cada etapa del diseño de diagramas de componentes de software:

3.1.4.1 Levantamiento

- **Responsables (R):** Los analistas de sistemas y en menor grado los arquitectos son los encargados de llevar a cabo el levantamiento de requisitos y asegurar que las necesidades del usuario sean correctamente entendidas y documentadas.
- **Aprueban (A):** Los stakeholders proporcionan la visión estratégica del proyecto y definen las restricciones operativas.
- **Consultados (C):** El equipo de desarrollo colabora en la revisión técnica de los requisitos y en la identificación de limitaciones.
- **Informados (I):** El equipo de calidad es informado sobre los resultados del levantamiento para utilizarlos en las fases siguientes.

3.1.4.2 Análisis

- **Responsables (R):** El arquitecto de software lidera el análisis de los requisitos levantados, validando la factibilidad técnica y organizando los componentes clave.
- **Aprueban (A):** El arquitecto de software tiene la responsabilidad final del resultado del análisis de la solución.
- **Consultados (C):** Los equipos de desarrollo, DBA y operaciones son consultados para revisar la

escalabilidad y viabilidad técnica y operativa. También es consultado el analista en caso de ser necesario aterrizar algún requerimiento, y en algunos casos suele ser necesario consultar con el stakeholder para validar temas no contemplados en la etapa de levantamiento.

- **Informados (I):** El equipo de desarrollo es informado sobre los avances en el análisis de la solución.

3.1.4.3 Diseño

- **Responsables (R):** El arquitecto de software es responsable de la elaboración y documentación del diagrama de componentes, asegurando que esté alineado con los resultados del análisis.
- **Aprueban (A):** El arquitecto de software es el responsable final del resultado de la etapa de diseño.
- **Consultados (C):** Los equipos de desarrollo, DBA y de operaciones son consultados para revisar las dependencias de la base de datos y la viabilidad técnica del diseño, así como la identificación de alguna limitante técnica.
- **Informados (I):** Los equipos de desarrollo, DBA y de operaciones son informados sobre la finalización del diseño.

3.1.4.4 Socialización

- **Responsables (R):** El arquitecto de software y los líderes técnicos son responsables de presentar el diagrama a los equipos clave.

- **Aprueban (A):** El arquitecto de software es el responsable final de la culminación exitosa de esta etapa.
- **Consultados (C):** Los equipos de DBA y operaciones son consultados para confirmar la viabilidad técnica del diseño.
- **Informados (I):** Los analistas, así como los equipos de calidad, DBA y operaciones son informados en la etapa de socialización.

Roles	Actividades			
	Levantamiento	Análisis	Diseño	Socialización
StakeHolder	A	C		
Analista	R	C		I
Arquitecto	R	R, A	R, A	R, A
Equipo Dev	C	C, I	C, I	R
Calidad	I			I
DBA		C	C, I	C, I
Operaciones		C	C, I	C, I

Tabla 3: Matriz RACI

Fuente: El autor

3.2 Encuestas y entrevistas

3.2.1 Encuestas

Respecto a la estrategia de encuesta, el formato puede ser revisado en la sección de anexos, y los temas que se evaluaron son los siguientes:

1. Perspectiva del arquitecto, respecto al grado de automatización que tiene actualmente el procedimiento para el diseño de diagramas de componentes para una solución de software.

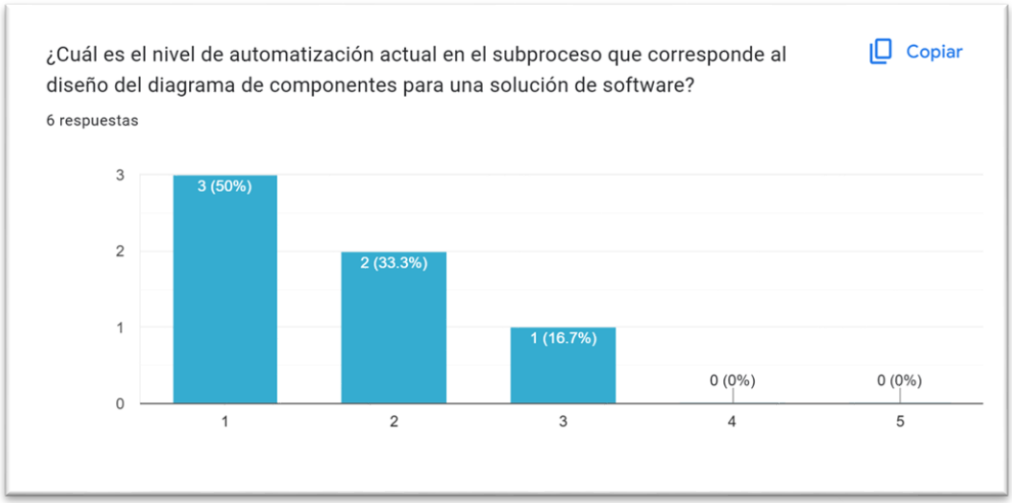


Figura III.4: Percepción del nivel de automatización del subproceso

Fuente: El autor

En la figura 3.4, se puede evidenciar que el usuario confirma la falta de automatización en el subproceso

2. Perspectiva del arquitecto, respecto a que tan dificultoso le resulta revisar, validar y analizar si una funcionalidad necesaria para el diseño de la solución ya existe en el sistema o debe ser definida como nueva.

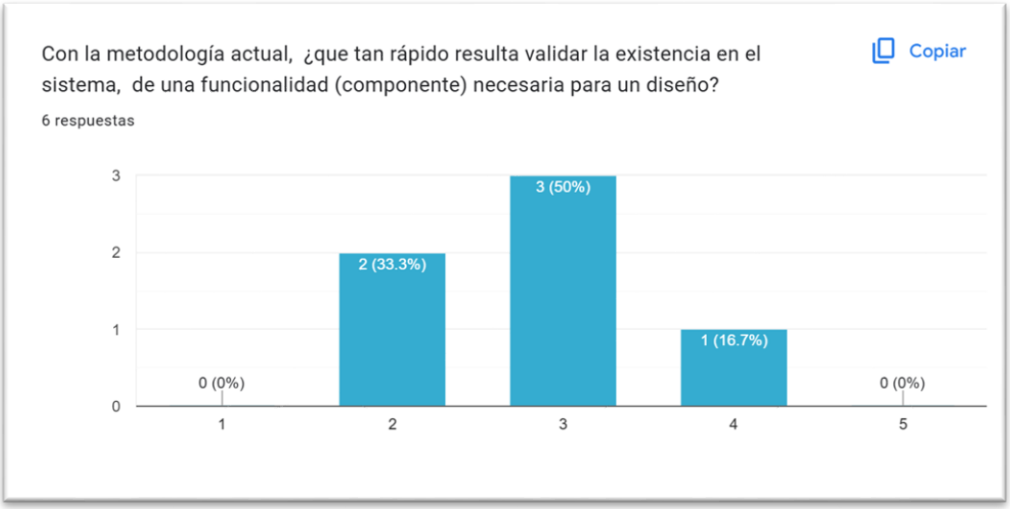


Figura III.5: Percepción sobre la validación de componentes reutilizables

Fuente: El autor

En la figura 3.5, se puede apreciar una ligera percepción hacia la dificultad en la validación de los componentes existentes

3. Perspectiva del arquitecto, respecto a cuánto tiempo en horas, le toma diseñar un diagrama de componentes para una solución de software en la cual se requiere la definición arquitectónica integral para la solución.

Figura III.6: Tiempo para elaboración de diagramas

Fuente: El autor

En la figura 3.6, se puede evidenciar que el tiempo medio para la elaboración de un diagrama de componentes es de alrededor de 19 horas.

4. Perspectiva del arquitecto, respecto a que tan fácil le resulta realizar ajustes en el diagrama de componentes una vez terminado.

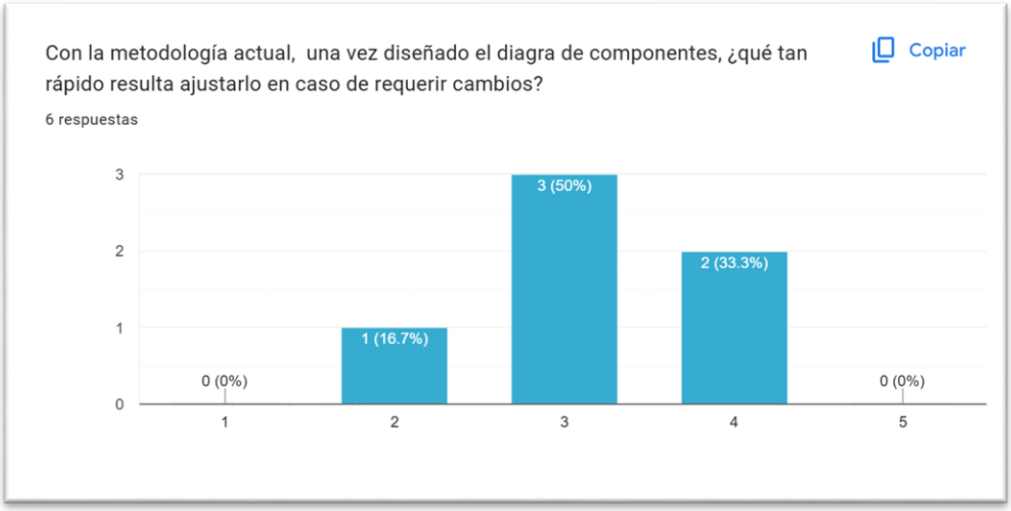


Figura III.7: Facilidad de realizar ajustes sobre diagramas terminados

Fuente: El autor

En la figura 3.7, se puede evidenciar una ligera percepción de facilidad al realizar ajustes sobre un diseño ya terminado

5. Perspectiva del arquitecto, respecto a que tan fácil es realizar el versionamiento de los diagramas de arquitectura.

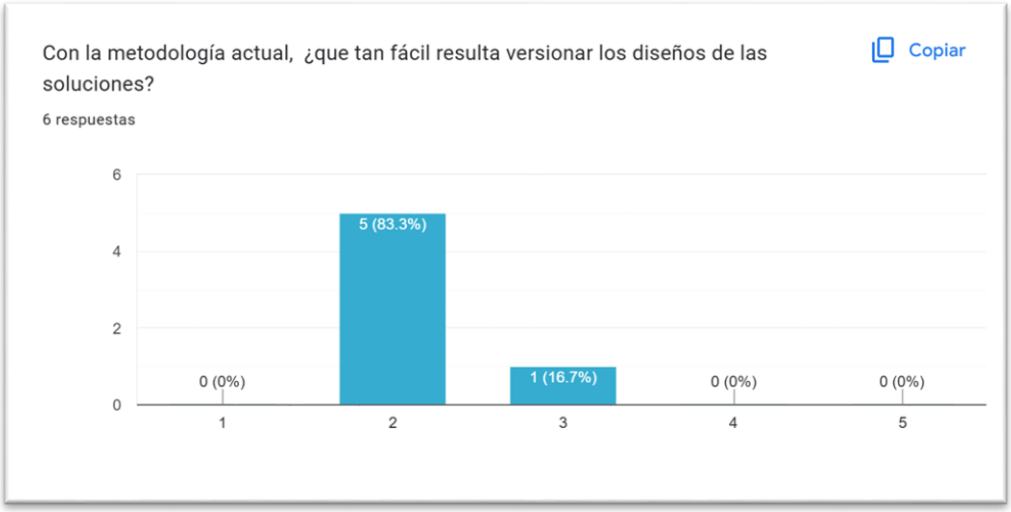


Figura III.8: Facilidad de versionamiento para los diseños

Fuente: El autor

En la figura 3.8, se puede evidenciar la dificultad en el versionamiento de los diseños

6. La última pregunta de la encuesta gira en torno a la variable de interés definida en el capítulo 1 de este proyecto, la cual intenta determinar el grado de aceptación por parte de los arquitectos, para una herramienta que genere diagramas base de componentes de software de forma automática, mediante el ingreso de los requisitos de la solución.

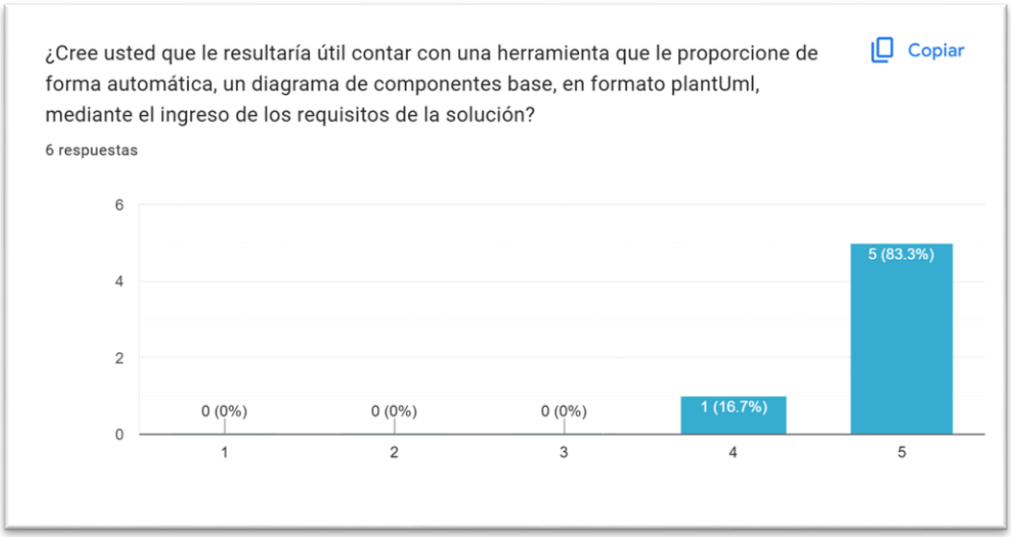


Figura III.9: Grado de aceptación de la propuesta

Fuente: El autor

En la figura 3.9, se puede evidenciar la aceptación favorable por parte de los arquitectos respecto a esta propuesta.

3.2.2 Entrevistas

Respecto a la entrevista realizada, se indagó sobre los siguientes puntos:

3.2.2.1 Identificación de las necesidades actuales

Se trataron varias necesidades actuales, entre las cuales destacan:

- El tiempo que toma diseñar los diagramas de componentes.
- El riesgo de incurrir en omisiones al analizar la existencia de soluciones actuales que suplan alguna necesidad en el nuevo diseño.
- La dificultad de realizar versionamiento de los diseños con las herramientas actuales.

- El riesgo latente en la seguridad de la información al utilizar herramientas en la nube.

3.2.2.2 Características de la solución propuesta

Se explicó y revisó las características principales de la solución propuesta, tales como:

- El hecho de ser un desarrollo propio y funcionar de forma local.
- El uso de una herramienta de modelado basado en texto como PlantUml.
- El uso de Inteligencia artificial para la generación de los diagramas.

3.2.2.3 Integración con herramientas existentes

Los arquitectos indicaron la necesidad de que esta propuesta de solución considere la integración con alguna de sus herramientas actualmente utilizadas, tales como: gitlab, para el versionamiento de los diseños.

3.2.2.4 Seguridad y privacidad

Los arquitectos resaltaron la importancia de la seguridad, sobre todo la importancia de proteger la información al momento de compartir datos sensibles con un LLM, como por ejemplo la base de componentes existentes.

3.2.2.5 Escalabilidad y capacidad de evolución

Se conversó y se acordó que la solución propuesta debería considerar a futuro poder generar todos los tipos de diagramas utilizados por los arquitectos al diseñar sus soluciones, así como poder integrarse con Gitlab para el versionamiento de los diseños.

3.3 Métricas

Debido a que el alcance del proyecto se enfoca en la etapa de diseño, no se podrá confirmar métricas objetivo, sin embargo, se pueden determinar ahora y ser validadas en una siguiente etapa de implementación del proyecto. Estas métricas se detallan a continuación en la tabla 4:

Indicador	Unidad	Actual	Objetivo
Tiempo necesario para generar un diagrama de componentes para una solución completa de arquitectura	Horas	19	2

Tabla 4: Métricas del proceso

Fuente: El autor

3.4 Limitaciones del proceso actual

Según la encuesta y la entrevista realizadas, se evidencias las siguientes limitaciones:

- Tiempo y esfuerzo manual
- Omisiones involuntarias
- Limitaciones en el versionamiento

3.5 Conclusiones

Es necesario que la propuesta de solución considere y brinde una alternativa de valor que mejore de forma significativa las limitaciones actuales del proceso.

CAPÍTULO IV

ANÁLISIS Y DISEÑO DE LA HERRAMIENTA PROPUESTA

El objetivo de este capítulo es realizar un análisis con base en la información obtenida en el capítulo anterior y elaborar el diseño de la solución cuyo alcance se enfocará en el diseño de los diagramas de componentes, para lo cual se desarrollará un prototipo funcional. Este prototipo permitirá visualizar de manera más tangible el valor que dicha solución puede aportar a la empresa objetivo del proyecto.

4.1 Análisis de la solución

Después de analizar el estado actual del proceso de diseño software que realiza la división de arquitectura de la empresa objetivo, tras llevar a cabo un minucioso levantamiento de información con los arquitectos y, tomando como base todo el conocimiento adquirido durante la investigación para la elaboración del marco teórico de este proyecto, se propone llevar a cabo el diseño de la solución con apoyo de las siguientes herramientas y tecnologías:

4.2 Herramientas y tecnologías

4.2.1 Herramienta de modelado UML

Se propone **PlantUml** como herramienta de modelado UML para este proyecto por su enfoque basado en texto, que facilita la integración con sistemas de control de versiones y fomenta una colaboración eficiente entre el equipo. Su capacidad para integrarse con modelos de lenguaje grandes (LLM) permite generar automáticamente descripciones y diagramas a partir de texto natural, optimizando la documentación. PlantUml soporta una amplia variedad de diagramas UML, agilizando el diseño y la documentación del sistema de manera rápida y sencilla. Además, al ser de código abierto, ofrece flexibilidad y personalización para adaptarse a las necesidades específicas del proyecto sin costos adicionales. Estas características hacen de PlantUml una opción robusta y eficiente, mejorando la calidad y cohesión en el desarrollo del proyecto.

4.2.2 Servicio de generación de diagrama

Se propone PlantUml Server como herramienta de generación de diagramas UML para este proyecto debido a su capacidad de operar de manera on premise, lo que garantiza un control total sobre los datos y la seguridad de la información sensible del proyecto. Al implementar PlantUml Server localmente, se puede integrar de forma eficiente la generación de diagramas UML dentro de la infraestructura existente, facilitando la automatización y la consistencia en la documentación técnica.

4.2.3 Large Language model

Se propone LLAMA 3.1 70b por ser un modelo open-source y altamente optimizable, que permite personalización completa y acceso a tecnologías avanzadas como cuantización para mejorar el rendimiento. Su capacidad para procesar hasta 128,000 tokens en múltiples idiomas, lo hace ideal para aplicaciones que

requieren manejo de texto extenso, mientras que su eficiencia en el uso de recursos lo convierte en una opción poderosa y flexible para este proyecto.

4.2.3.1 Técnica para mejorar la precisión de las respuestas

Se propone RAG para potenciar el modelo de lenguaje debido a su capacidad única de combinar generación de texto con recuperación de información relevante, enriqueciendo la comprensión contextual al integrar información específica, lo que resulta en respuestas más coherentes y fundamentadas. Al reducir las alucinaciones del modelo, se garantiza mayor fiabilidad en los resultados.

4.2.3.2 Hardware para procesamiento del LLM

El gran modelo de lenguaje seleccionado se ejecutará sobre servidores DGX con GPUs H100. Estos servidores ya existen en el ecosistema tecnológico de la empresa objetivo de este proyecto.

4.2.4 Bases de datos

4.2.4.1 Milvus Db

Se propone **Milvus** como base de datos para gestionar embeddings en este proyecto por su diseño especializado en almacenar y buscar vectores de alta dimensión de manera eficiente. Ofrece un rendimiento sobresaliente en búsquedas de similitud. Además, al ser de código abierto, Milvus proporciona flexibilidad y personalización para adaptarse a las necesidades futuras del proyecto sin costos adicionales. Estas características hacen de Milvus una opción robusta y eficiente para la gestión de embeddings, asegurando un rendimiento óptimo y una escalabilidad sostenida.

4.2.4.2 Mongo Db

Se propone MongoDB como la base de datos para gestionar la metadata de la aplicación debido a su flexibilidad en el manejo de esquemas dinámicos, lo que permite almacenar datos estructurados y semi-estructurados de manera eficiente. Esta característica es especialmente útil para la metadata, que a menudo varía en estructura y puede evolucionar con el tiempo. Otro aspecto fundamental que respalda la elección de MongoDB es su capacidad de escalabilidad horizontal y su alto rendimiento en operaciones de lectura y escritura. Esto asegura que la gestión de metadata pueda manejar grandes volúmenes de datos y crecer junto con las necesidades de la aplicación sin comprometer la eficiencia.

4.2.5 Lenguajes de programación

4.2.5.1 Java

Se propone Java debido a su robustez, escalabilidad y amplio soporte en la industria. Java es reconocido por su capacidad para manejar aplicaciones de gran escala con eficiencia, lo que garantiza que el Backend pueda crecer y adaptarse a las demandas crecientes de los sistemas sin comprometer el rendimiento. Además, su ecosistema maduro, que incluye frameworks como Spring Boot, facilita el desarrollo rápido y estructurado, permitiendo implementar funcionalidades complejas de manera eficiente y mantenible.

4.2.5.2 Python

Se propone Python como el lenguaje para la lógica de RAG (Retrieval-Augmented Generation) en este proyecto debido a sus excepcionales capacidades para el desarrollo de aplicaciones de inteligencia artificial y aprendizaje automático. Python cuenta con un ecosistema robusto de bibliotecas y frameworks especializados, como TensorFlow, PyTorch y Hugging Face, que simplifican la implementación, entrenamiento y despliegue de modelos avanzados de IA. Su sintaxis clara y concisa permite un desarrollo ágil y eficiente, facilitando la experimentación y optimización de algoritmos complejos necesarios para la lógica de RAG.

4.2.5.3 TypeScript

Se propone TypeScript debido a su capacidad para mejorar la calidad y la mantenibilidad del código en proyectos de gran escala. TypeScript, al ser un superconjunto tipado de JavaScript, ofrece ventajas significativas en términos de detección temprana de errores y autocompletado inteligente, lo que reduce significativamente los bugs y mejora la eficiencia del desarrollo. Además, TypeScript se integra perfectamente con modernos frameworks de FrontEnd como Angular, permitiendo construir interfaces de usuario dinámicas y responsivas con mayor facilidad y seguridad.

4.2.6 Framework para FrontEnd

Se propone Angular como el framework de FrontEnd para este proyecto debido a su sólida integración con TypeScript, lo que

garantiza un desarrollo más estructurado y tipado, reduciendo errores y mejorando la mantenibilidad del código. Angular ofrece una arquitectura robusta basada en componentes, lo que facilita la creación de interfaces de usuario escalables y reutilizables. Además, su sistema de inyección de dependencias y su enrutador avanzado permiten gestionar de manera eficiente la complejidad de aplicaciones de gran envergadura, asegurando una experiencia de desarrollo fluida y organizada.

4.2.7 Framework para Backend

Se propone Spring Boot como el framework para este proyecto, debido a su capacidad para simplificar y acelerar el desarrollo de aplicaciones Java robustas y escalables. Su enfoque de convenciones sobre configuraciones reduce el tiempo de desarrollo y minimiza errores, mejorando la eficiencia. La amplia comunidad y el soporte continuo garantizan actualizaciones constantes y acceso a numerosos recursos, asegurando un Backend sólido, mantenible y preparado para futuras expansiones del proyecto.

4.2.8 Tecnología de contenedorización

Se propone Docker como la herramienta de contenedorización para este proyecto, debido a su capacidad para crear entornos consistentes y portables, asegurando un funcionamiento uniforme en desarrollo, prueba y producción. Docker simplifica la gestión de dependencias y el despliegue, reduciendo conflictos y mejorando la eficiencia de recursos gracias a su arquitectura ligera. Además, es compatible con Kubernetes, lo que facilita la orquestación y escalabilidad de los contenedores en entornos de producción. Su robusto ecosistema, que incluye herramientas

como Docker Compose y Swarm, optimiza la automatización de contenedores.

4.2.9 Herramienta de autenticación única

Se propone CAS (Central Authentication Service) por su capacidad para centralizar la autenticación, mejorando la seguridad y la gestión de usuarios en toda la aplicación. CAS soporta múltiples protocolos, lo que facilita su integración con diversas aplicaciones y servicios existentes. Además, al ser una solución de código abierto, ofrece flexibilidad y personalización sin costos adicionales, adaptándose a las necesidades futuras del proyecto. Su arquitectura escalable y el respaldo de una comunidad activa aseguran un rendimiento óptimo y actualizaciones constantes, haciendo de CAS una opción confiable para proporcionar una experiencia de usuario fluida y segura.

4.2.10 Herramienta para balanceo de carga

Se propone Traefik como herramienta para el balanceo de carga debido a su diseño moderno y su excelente compatibilidad con entornos de microservicios. Traefik se integra fácilmente con Docker, lo que facilita la gestión dinámica de las rutas y la detección automática de servicios. Su capacidad para manejar configuraciones en tiempo real permite una escalabilidad eficiente y una respuesta rápida a los cambios en la infraestructura. Al ser una solución de código abierto respaldada por una comunidad activa, Traefik proporciona flexibilidad y soporte continuo, adaptándose perfectamente a las necesidades futuras del proyecto.

4.3 Arquitectura de la solución

4.3.1 Nuevo flujo (TO-BE) para el diseño de diagramas de componentes

Dado que el objetivo de este proyecto es automatizar la creación de diagramas de componentes, que constituye solo una parte del proceso de diseño de una solución de software, a continuación, se describe el nuevo flujo establecido para esta fase del diseño:

1. El usuario accede a la aplicación, en la cual deberá ingresar los criterios o requisitos para la generación del diagrama de componentes, así como el nombre para dicho diagrama.
2. Una vez definidos los criterios de creación, se presiona el botón de procesamiento y se dispara la petición hacia el Backend de la aplicación.
3. La aplicación recibe la petición y obtiene información de contexto a partir de los requisitos de creación.
4. Con la información de contexto y los requisitos de creación se elabora un prompt que será el insumo para el LLM.
5. Se realiza la petición al LLM para la generación del diagrama.
6. El LLM procesa la solicitud y devuelve el resultado.
7. La aplicación recibe la respuesta del LLM y la devuelve a la capa FrontEnd.
8. La capa FrontEnd vuelve a enviar una petición al Backend para solicitar el renderizado del diagrama.
9. El Backend solicitad el renderizado al servidor PlantUml.
10. PlantUml Server realiza el renderizado y devuelve el resultado a la aplicación.
11. La aplicación devuelve a la capa FrontEnd el diagrama renderizado.
12. El usuario recibe el resultado y analiza si es que este cumple con sus expectativas.

13. Si el usuario está conforme con el resultado, finaliza el proceso y no se cumplen los pasos 14 y 15.
14. Si el usuario no está conforme con el resultado, tiene dos alternativas:
    a. Puede realizar los ajustes el mismo en la pantalla de la aplicación.
    b. Puede ingresar de forma iterativa, requisitos de ajuste y solicitar a la aplicación que se modifique el diagrama según esos nuevos requisitos y tomando como base el diagrama actual
15. Retoma el flujo desde el paso 8.

En la Figura 4.1 se puede apreciar el nuevo flujo correspondiente al diseño de diagramas de compontes.

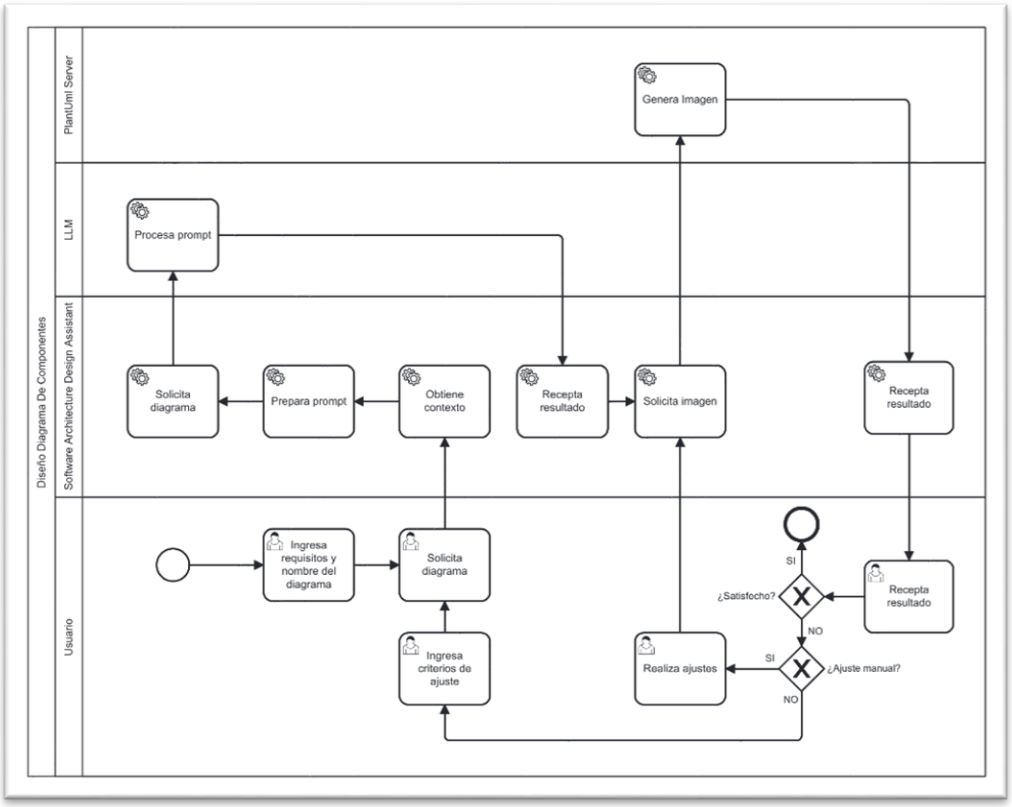


Figura IV.1: Modelo TO-BE para diseño de diagrama de componentes

Fuente: El autor

4.3.2 Diagrama de contexto

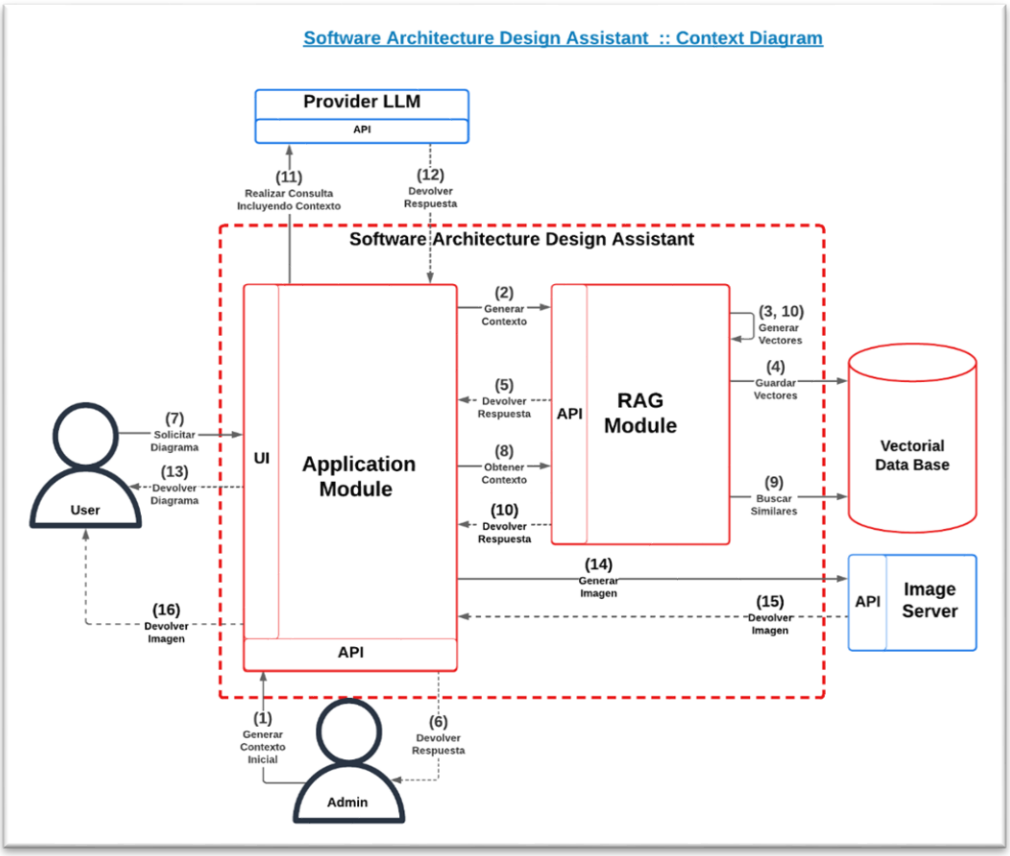


Figura IV.2: Diagrama de contexto de la solución

Fuente: El autor

4.3.3 Casos de uso

Basándose en el diagrama de contexto y de acuerdo con el alcance definido, se identifican los dos siguientes casos de uso:

4.3.3.1 Generación manual de información de contexto

1 Un usuario admin, accede a un cliente Rest, por ejemplo “Postman”.

2 El usuario realiza una petición a la aplicación, enviando un listado de diccionarios de datos para generar información de contexto.

3 La aplicación redirecciona esta solicitud al módulo de RAG.
4 El módulo de RAG realiza las siguientes acciones para cada uno de los ítems del listado de diccionarios.
    a. Genera un vector para dicha información.
    b. Almacena el vector en la base Milvus.
5 El módulo de RAG devuelve el resultado.
6 La aplicación devuelve el resultado

4.3.3.2 Generación de diagrama de componentes

Este caso de uso se detalla en la sección 4.3.1

4.3.4 Diagrama de componentes

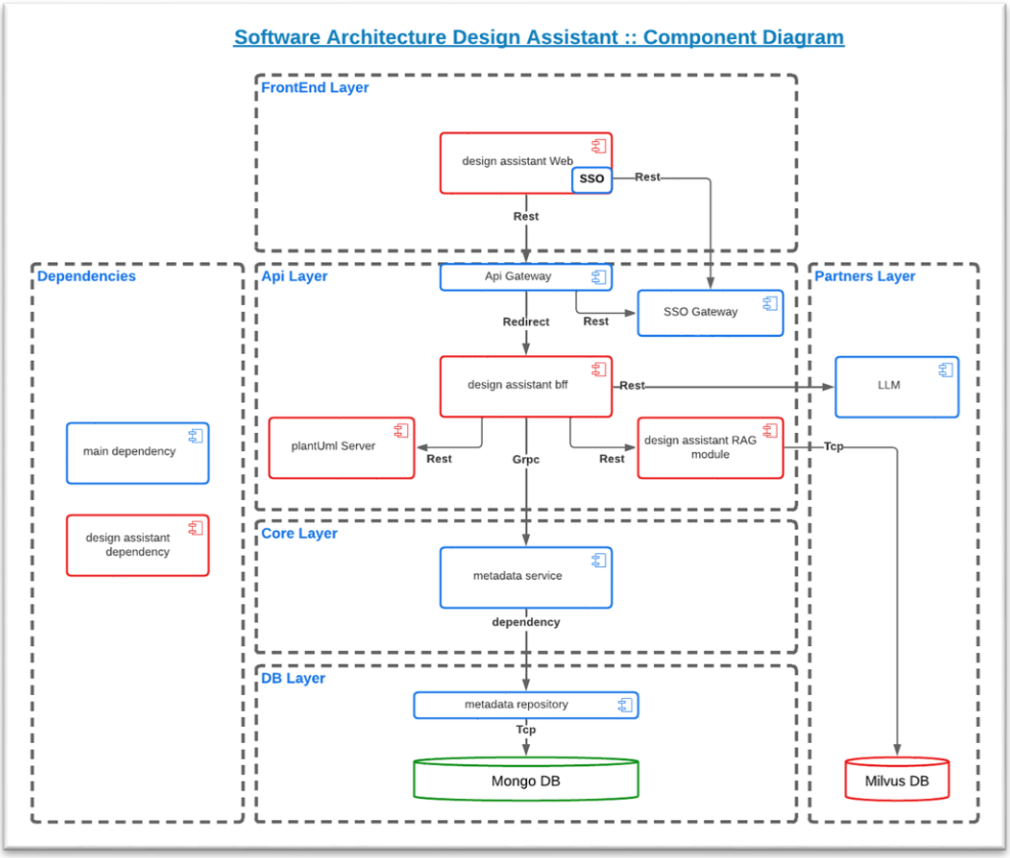


Figura IV.3: Diagrama de componentes de la solución

Fuente: El autor

4.3.5 Diagramas de secuencias

4.3.5.1 Generación manual de información de contexto

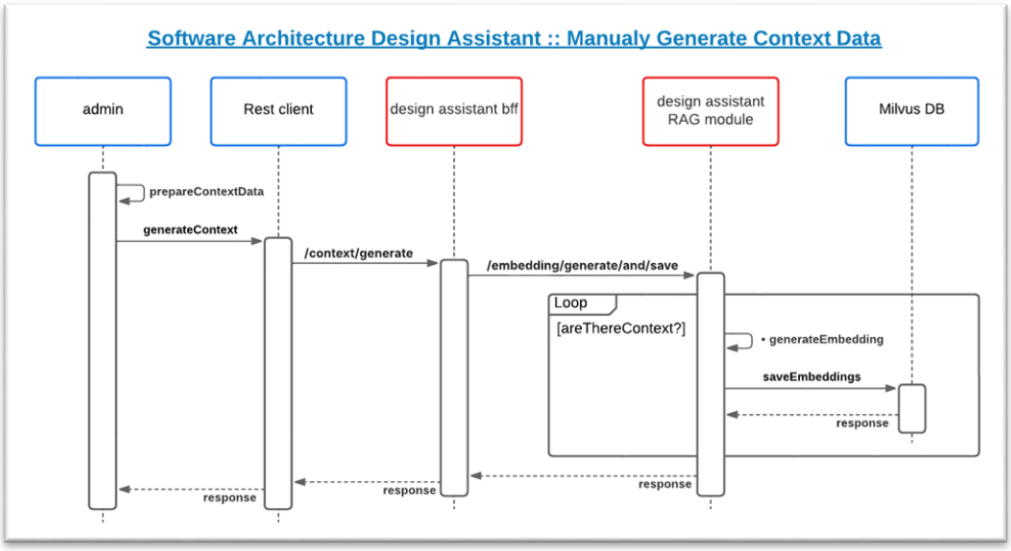


Figura IV.4: Generación manual de contexto

Fuente: El autor

4.3.5.2 Generación de diagrama de componentes

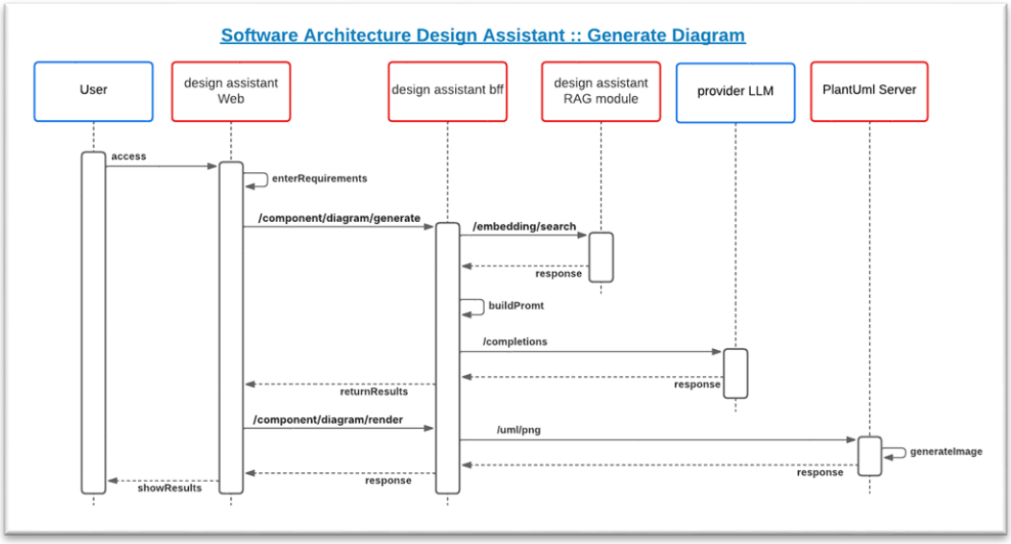


Figura IV.5: Generación de diagrama de componentes

Fuente: El autor

4.3.6 Diagrama de clases

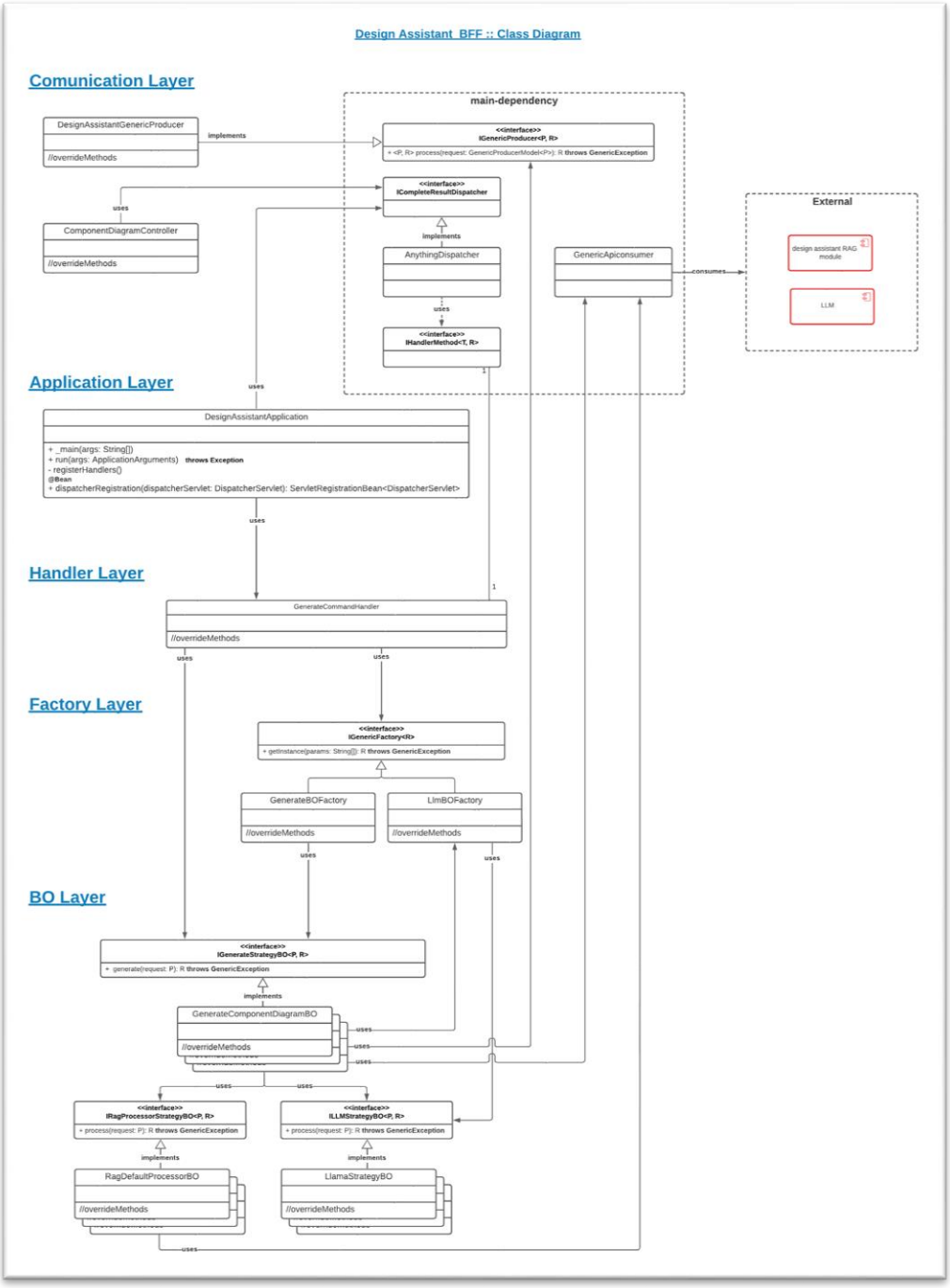


Figura IV.6: Diagrama de clases

Fuente: El autor

En la figura 4.6 correspondiente al diagrama de clases del componente SpringBoot, se puede apreciar el uso de principios y patrones para lograr un diseño robusto y extensible. Se puede apreciar la aplicación de los principios SOLID, así como el uso de varios patrones tales como: Abstract Factory, Strategy, Bridge, Mediator, Facade, etc.

4.4 Desarrollo del prototipo

4.4.1 Introducción

Se desarrolló un prototipo funcional para que el usuario pueda validar de manera tangible los beneficios que esta herramienta ofrece. El prototipo intenta cumplir en la mayor medida posible con todo el stack tecnológico definido en la sección 4.2; sin embargo, es importante considerar que su alcance es limitado al ser solo un prototipo. Bajo esa premisa, a continuación, se presenta la interfaz de usuario del prototipo desarrollado:

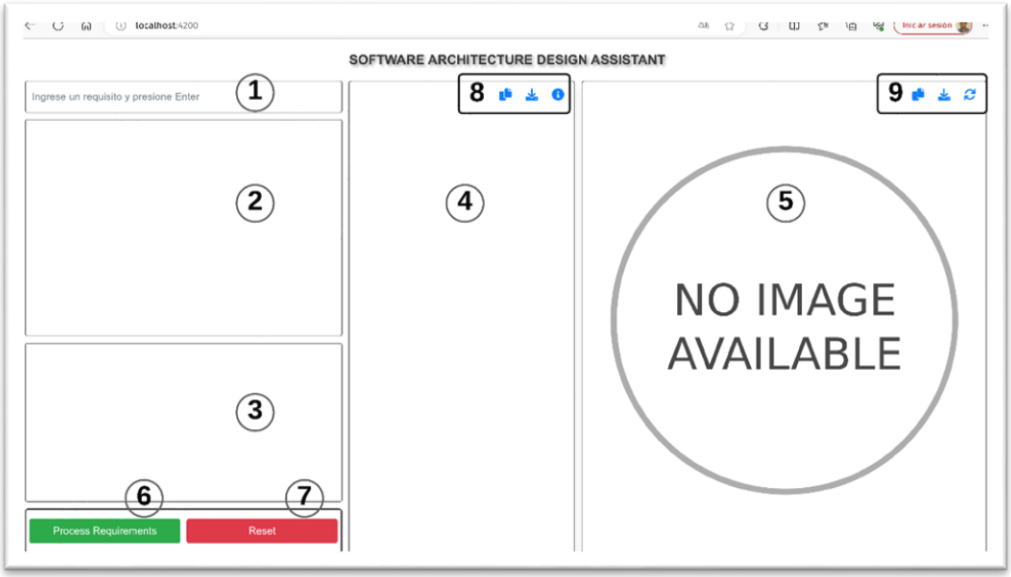


Figura IV.7: Interfaz de usuario

Fuente: El autor

En la figura 4.7 se puede validar las diferentes secciones con las que cuenta la interfaz de usuario, tales como:

1. Sección para el ingreso de requisitos.
2. Sección para visualización y gestión de requisitos ingresados.
3. Sección de historial de requisitos ingresados.
4. Sección de visualización y edición de código PlantUml.
5. Sección de visualización de diagrama renderizado.
6. Botón para solicitar la generación del diagrama.
7. Botón para limpiar el formulario.
8. Opciones para copiar, descargar y ver descripción del código.
9. Opción para copiar, descargar y refrescar el diagrama según el contenido del código.

4.4.2 Generación de información de contexto

4.4.2.1 Descripción de la prueba

Para esta prueba se construyó un diccionario de componentes ficticio para poder alimentar la base de datos vectorial y que dicha información sirva como de contexto para la prueba de generación de los diagramas de componentes. La estructura del diccionario puede ser revisada en la sección de anexos:

4.4.2.2 Contrato

El detalle de los contratos se puede ver en la sección de anexos.

4.4.2.3 Consumo

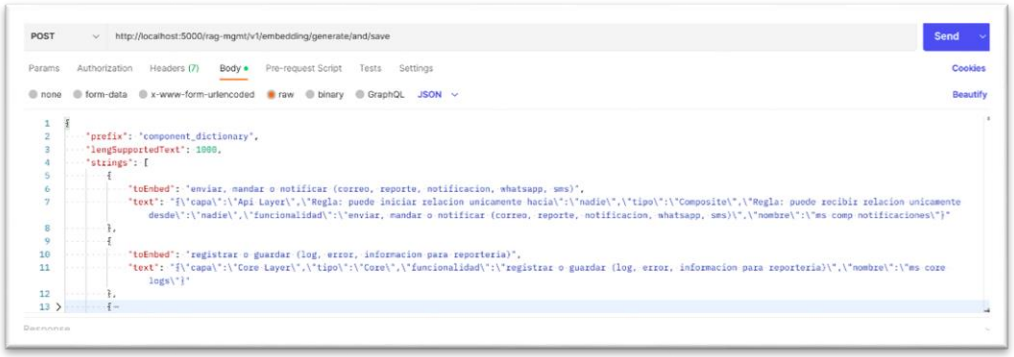


Figura IV.8: Consumo para generar contexto

Fuente: El autor

4.4.2.4 Respuesta

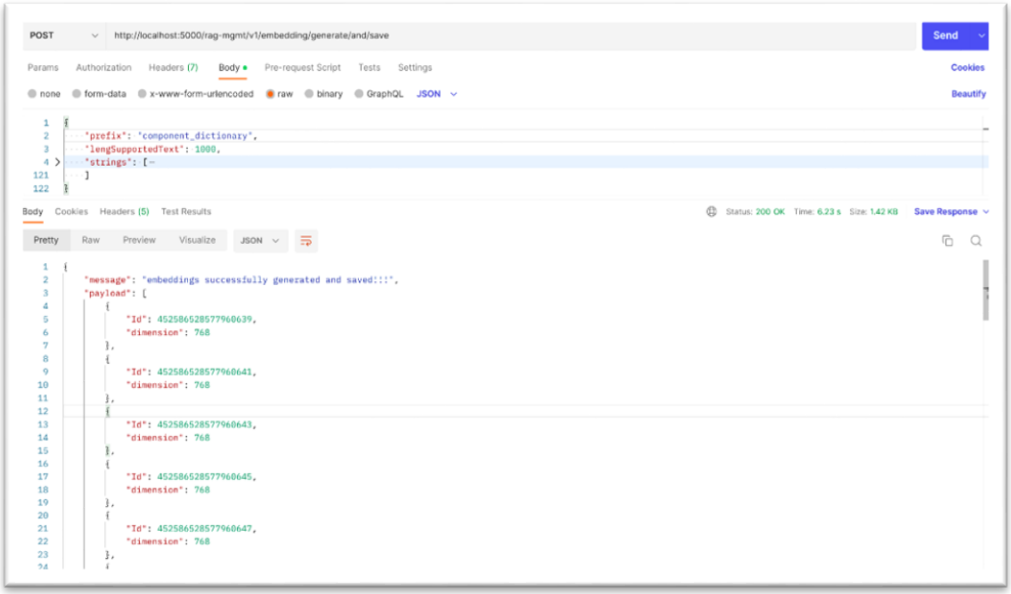


Figura IV.9: Resultado de generación de contexto

Fuente: El autor

4.4.3 Generación de diagrama de componentes

4.4.3.1 Descripción de la prueba

Para la generación del diagrama de componentes se consideró una solución ficticia basada en los requisitos que se detallan a continuación:

4.4.3.2 Requisitos de prueba

- El objetivo de esta solución es el procesamiento de pagos de clientes.
- Los pagos deben ser ingresados desde la aplicación que utilizan los clientes.
- Se debe hacer uso de reglas de negocio con camunda para validar acciones necesarias según el saldo pendiente.
- Se debe enviar por correo una notificación de confirmación al cliente, al recibir un pago.
- Luego de cada pago recibido se debe realizar la actualización del del saldo del cliente.
- Se debe realizar el registro de log de errores de forma asíncrona.

4.4.3.3 Ejecución de la prueba

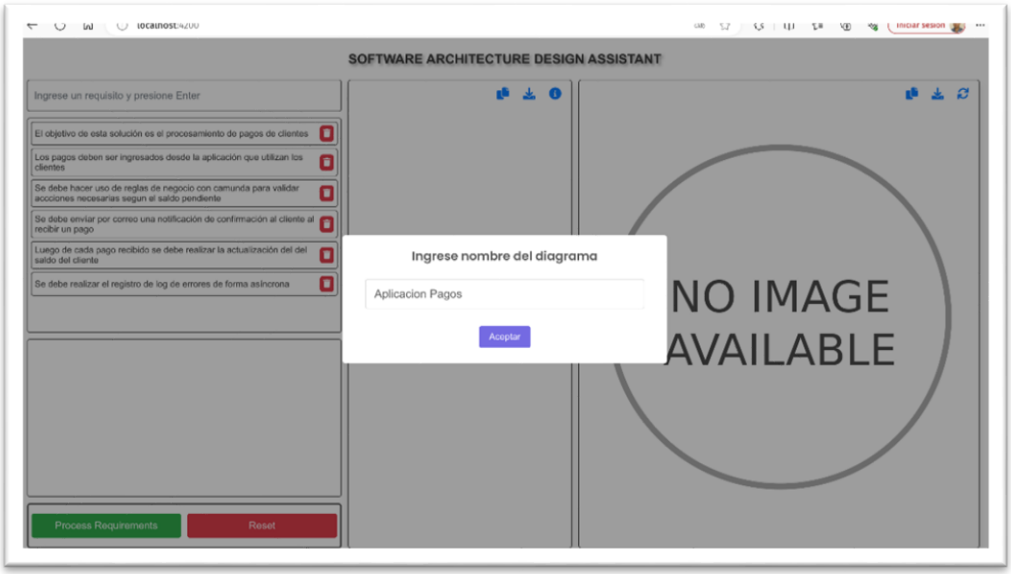


Figura IV.10. Ejecución de solicitud de diagrama

Fuente: El autor

4.4.3.4 Resultado

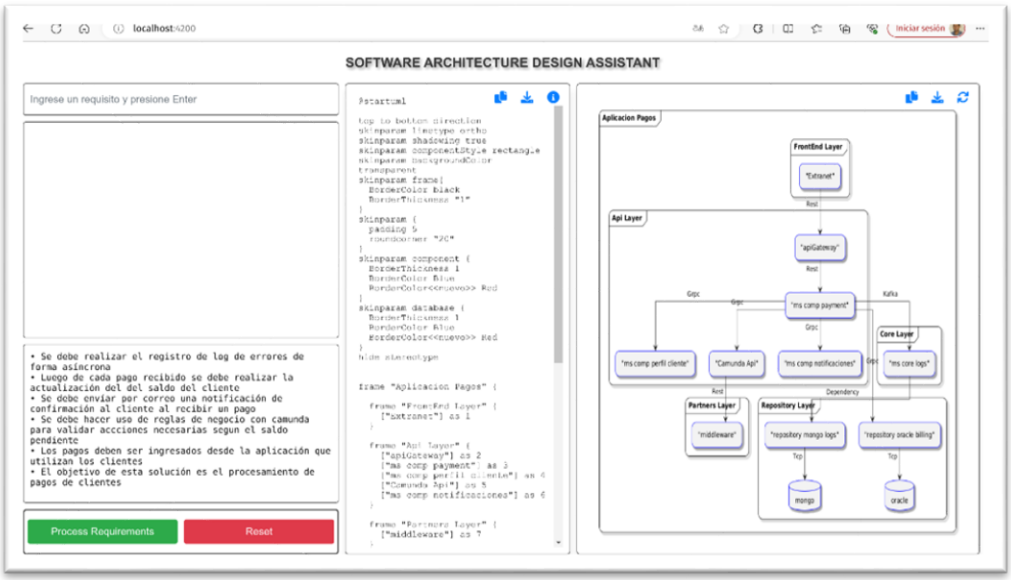


Figura IV.11: Resultado de generación de diagrama

Fuente: El autor

Luego de la ejecución se obtuvo los siguientes resultados:

- **Diagrama en formato PlantUml**
  El diagrama en formato PlantUml se puede apreciar en la sección de anexos.
- **Diagrama en formato png**
  El diagrama en formato Imagen se puede apreciar en la sección de anexos.

CAPÍTULO V

EVALUACIÓN Y ANÁLISIS DE RESULTADOS

En este capítulo se lleva a cabo la revisión de los resultados de la validación del prototipo con los usuarios. El objetivo principal es interpretar dichos resultados y determinar su relevancia en el contexto del proyecto. Este capítulo servirá como base para las conclusiones finales y las recomendaciones futuras, destacando cómo los resultados contribuyen al avance del conocimiento en este tema.

5.1 Validación del prototipo con el usuario

Se realizó la revisión del prototipo con los arquitectos en dos sesiones, en las cuales se revisó la interfaz, sus funcionalidades y se hicieron pruebas de generación de diagramas.

5.2 Elaboración y toma de encuestas

Posteriormente se preparó una nueva encuesta, con la finalidad de validar el grado de aceptación de los arquitectos respecto al prototipo evaluado. El formato completo de la encuesta se puede revisar en la sección de anexos, y los temas que se consultaron son los siguientes:

1. Se solicita al usuario, que califique en una escala del 1 al 5, que tan intuitiva le pareció la interfaz del prototipo. Siendo 1 muy poco intuitiva y 5 muy intuitiva.

Dirección de correo electrónico	¿Qué tan intuitiva te pareció la interfaz del prototipo?
dlino@telconet.ec	4
cocedeno@gmail.com	4
arsuarez@telconet.ec	4
jlung@telconet.ec	3
jdvinueza@telconet.ec	4
cxcastro@telconet.ec	5

Figura V.1: Percepción sobre la usabilidad del prototipo

Fuente: El autor

En la figura 5.1, correspondiente a la primera pregunta de la encuesta, los resultados evidencian respuestas mayormente favorables acerca de que tan intuitivo es el prototipo desarrollado.

2. Se solicita al usuario que indique, calificando en una escala del 1 al 5, que tanto el prototipo le facilitó la generación de los diagramas de componentes, respecto a su metodología actual. Siendo 1 nada fácil y 5 muy fácil.

Dirección de correo electrónico	¿El prototipo facilitó la generación de diagramas de componentes en comparación con tus métodos habituales?
dlino@telconet.ec	5
cocedeno@gmail.com	5
arsuarez@telconet.ec	5
jlung@telconet.ec	4
jdvinueza@telconet.ec	4
cxcastro@telconet.ec	4

Figura V.2: Facilitar el diseño de diagramas

Fuente: El autor

En la figura 5.2 correspondiente a la segunda pregunta de la encuesta, se evidencia una alta aprobación, respecto a que tanto el prototipo facilita la generación de diagramas de componentes.

3. Se solicita al usuario que indique algún problema técnico que haya notado durante las pruebas con el prototipo.

Dirección de correo electrónico	¿Encontraste algún obstáculo técnico al utilizar el prototipo? Si es así, por favor indícalo.
dlino@telconet.ec	Ninguno
cocedeno@gmail.com	No por el momento
arsuarez@telconet.ec	Ninguna
jlung@telconet.ec	Ninguno
jdvinueza@telconet.ec	Ninguno
cxcastro@telconet.ec	Ninguno

Figura V.3: Dificultades técnicas con el prototipo

Fuente: El autor

En la figura 5.3 correspondiente a la tercera pregunta de la encuesta se verifica que los usuarios no encontraron observaciones técnicas respecto al uso del prototipo desarrollado.

4. Se solicita al usuario que califique en una escala del 1 al 5, la precisión de los diagramas de componentes generados con el prototipo. Siendo 1 nada preciso y 5 muy preciso.

Dirección de correo electrónico	¿Cómo calificarías la precisión de los diagramas generados por el prototipo?
dlino@telconet.ec	4
cocedeno@gmail.com	4
arsuarez@telconet.ec	3
jlung@telconet.ec	4
jdvinueza@telconet.ec	3
cxcastro@telconet.ec	4

Figura V.4: Precisión de los diagramas

Fuente: El autor

En la figura 5.4, correspondiente a la cuarta pregunta de la encuesta, se evidencia una calificación ligeramente por encima de la media, lo cual es comprensible, ya que se trata de diagramas base, los cuales pueden ser mejorados con base en iteraciones con la misa herramienta o con ajustes manuales del usuario. Adicionalmente la precisión de los diagramas puede mejorarse conforme se enriquezca más la base de contexto y se realicen ajustes sobre el prompt utilizado.

5. Se solicita al usuario que califique en una escala del 1 al 5, que tan rápido le pareció la generación de diagramas de componentes utilizando el prototipo. Siendo 1 nada rápido y 5 muy rápido.

Dirección de correo electrónico	¿Qué tan rápido es el proceso de generación de diagramas con el prototipo?
dlino@telconet.ec	5
cocedeno@gmail.com	5
arsuarez@telconet.ec	4
jlung@telconet.ec	4
jdvinueza@telconet.ec	3
cxcastro@telconet.ec	4

Figura V.5: Rapidez en la generación de diagramas

Fuente: El autor

En la figura 5.5, correspondiente a la quinta pregunta de la encuesta, se evidencia mayormente una buena percepción del usuario respecto a la rapidez del prototipo para generar los diagramas de componentes.

6. Se solicita al usuario que califique en una escala del 1 al 5, que tan útil le pareció la asistencia de un LLM en la generación de diagramas de componentes. Siendo 1 nada útil y 5 muy útil.

Dirección de correo electrónico	¿Te resultó útil la asistencia del LLM en la generación de diagramas?
dlino@telconet.ec	5
cocedeno@gmail.com	5
arsuarez@telconet.ec	5
jlung@telconet.ec	3
jdvinueza@telconet.ec	4
cxcastro@telconet.ec	5

Figura V.6: Utilidad en la asistencia de un LLM

Fuente: El Autor

En la figura 5.6, correspondiente a la sexta pregunta de la encuesta, se evidencia que la mayoría de los arquitectos consideran de mucha utilidad la asistencia de un LLM en la generación de diagramas de componentes.

7. Se solicita al usuario que califique en una escala del 1 al 5, cuál es su grado de satisfacción con el uso del prototipo. Siendo 1 nada satisfecho y 5 muy satisfecho.

Dirección de correo electrónico	¿Cómo evalúas tu satisfacción general con el prototipo?
dlino@telconet.ec	5
cocedeno@gmail.com	4
arsuarez@telconet.ec	4
jlung@telconet.ec	4
jdvinueza@telconet.ec	4
cxcastro@telconet.ec	5

Figura V.7: Satisfacción con el uso del prototipo

Fuente: El autor

En la figura 5.7, correspondiente a la séptima pregunta de la encuesta, se evidencia un aceptable nivel de satisfacción de los arquitectos, respecto al uso del prototipo.

8. Se solicita al usuario que califique con SI o NO, si considera que el uso de la herramienta podría mejorar la eficiencia en sus proyectos.

Dirección de correo electrónico	¿Consideras que el prototipo podría aumentar la eficiencia en tus proyectos?
dlino@telconet.ec	Si
cocedeno@gmail.com	Si
arsuarez@telconet.ec	Si
jlung@telconet.ec	Si
jdvinueza@telconet.ec	Si
cxcastro@telconet.ec	Si

Figura V.8: Incidencia del prototipo en la eficiencia de los diseños

Fuente: El autor

En la figura 5.8, correspondiente a la octava pregunta de la encuesta, se evidencia un rotundo acuerdo por parte de los arquitectos, respecto al aumento de la eficiencia en sus procesos de diseño, al contar con una herramienta de este tipo.

9. Se solicita el usuario que indique que aspectos de la herramienta le parecieron más valiosos.

Dirección de correo electrónico	¿Qué aspectos del prototipo te parecieron más valiosos?
dlino@telconet.ec	al momento de visualizar el diagrama con los datos de entrada
cocedeno@gmail.com	La generacion automatica de diagramas
arsuarez@telconet.ec	El uso del contexto relacionado a nuestro trabajo
jlung@telconet.ec	La integración con diferentes servidores de modelos LLM, internos y de terceros, aunque hay temas de seguridad que resolver
jdvinueza@telconet.ec	entrega un diagrama en base a os requeriientos entrgados bastante bueno.
cxcastro@telconet.ec	La generación de diagramas con una arquitectura basado en el requerimiento, permite partir con una idea que ayuda con el desarrollo.

Figura V.9: Aspectos destacados del prototipo

Fuente: El autor

En la figura 5.9, correspondiente a la novena pregunta de la encuesta, se visualiza los aspectos, que, a criterio de los arquitectos, son los más valiosos o destacados. A continuación de se detalla cada uno de ellos:

- Uno de los arquitectos resalta el hecho de que el prototipo permite tanto la visualización del diagrama en formato imagen, así como el diagrama en formato editable PlantUml y los criterios de entrada que permitieron la generación de este.
- Otro arquitecto destaca como tal la funcionalidad de generar de forma totalmente automática un diagrama de componentes completo, solo mediante el ingreso de algunos requisitos.
- Otro arquitecto señala como valioso, el hecho de que el diagrama generado, considere componentes ya existentes en el ecosistema tecnológico de la empresa, agregando un alto valor contextual al resultado.

- Aunque el objetivo del proyecto es consumir un LLM local, el prototipo cuenta con la capacidad de poder implementar el consumo al cualquier proveedor de LLM y hacer switch a cualquiera de ellos en tiempo de ejecución. Esta es la capacidad que resalta uno de los arquitectos. Sin embargo, el también hace énfasis en la importancia de la seguridad que se implemente en caso de utilizar dicha funcionalidad.
- Otro arquitecto destaca la calidad que presenta el diagrama de componentes generado con el prototipo a partir de unos cuantos requisitos ingresados.
- Finalmente, otro arquitecto señala la importancia de contar rápidamente con un diagrama base, a partir del cual se pueda iterar para mejorarlo, o tomarlo directamente como diagrama final, dependiendo de su exactitud.

10. Se solicita al usuario que indique, que funcionalidades adicionales considera que el prototipo debería incorporar.

Dirección de correo electrónico	¿Qué funcionalidades adicionales te gustaría que el prototipo incorporara para mejorar su integración?. Redáctalo por ítems
dlino@telconet.ec	seleccion de tipos de diagrama
cocedeno@gmail.com	otros tipos de diagramas UML
arsuarez@telconet.ec	**1.** Deberia poder generarse una especie de Workspace inicial y en ese trabajar por cada proyecto asignado y no estar pidiendo ingresar le nombre para descargar, la creacion del nombre deberia ser una funcionalidad del workspace. **2 .**Tener cuidado con el renderizado de la imagen, asi sea un preview, debe ser consistente. **3.** La descarga de la imagen que sea en PDF o jpg, , la transparencia del png no ayuda si se abre la imagen pura. **4.** Que en un futuro se integre con el repositorio corporativo.
jlung@telconet.ec	- Básicamente la capa de seguridad, necesitaría ser mejorada para gestionar reglas o enrutamiento de las solicitudes a los diferentes proveedores LLM
jdvinueza@telconet.ec	que haga commit del diagrama a un repositorio gitlab
cxcastro@telconet.ec	Permita importar e interpretar diagramas

Figura V.10 : Sugerencias de mejora

Fuente: El autor

En la figura 5.10, correspondiente a la décima pregunta de la encuesta, se visualiza varios ítems de mejora basados en el criterio y experiencia de cada uno de los arquitectos. A continuación, se detalla cada uno de ellos:

- Uno de los arquitectos, señala como punto de mejora, la posibilidad de que la herramienta permita seleccionar el tipo de diagrama que se desea generar. Esto implica que la herramienta implemente la gestión para otros tipos de diagramas UML con ayuda del LLM, lo cual no está contemplado en el alcance de este proyecto, pero es un excelente ítem a considerar para futuras fases.

- Si bien es cierto, el anterior arquitecto señala solo la necesidad de poder escoger el tipo de diagrama, otro arquitecto hace énfasis en contar con la implementación concreta para poder generar varios tipos de diagramas, lo cual ya fue indicado en el ítem anterior.
- Otra sugerencia se centra en la necesidad de contar con espacios de trabajo específicos por cada proyecto, de manera que no se solicite el ingreso del nombre del diagrama cada vez que se realiza una nueva iteración de mejora. Se aclaró con el arquitecto que al tratarse de un prototipo no cuenta con todas las optimizaciones que debería tener un aplicativo final, sin embargo, se considerará su propuesta como punto de mejora en futuras fases.
- Otra sugerencia recalca la importancia de la relación de aspecto de la imagen, de modo que, no se distorsione en caso de que sus dimensiones no coincidan con la forma del visualizador.
- Otra sugerencia enfatiza la necesidad de poder descargar la imagen como pdf, dado que el prototipo actualmente solo permite descargar la imagen en formato png.
- Otra sugerencia resalta la necesidad de que la herramienta pueda integrarse con el repositorio corporativo, de manera que pueda haber esta comunicación bidireccional y sea más ágil la replicación de los diseños en el repositorio corporativo.
- Otra sugerencia hace énfasis en el tema de seguridad, específicamente en el caso de uso en el cual se realice comunicación con un LLM distinto al local. Recalca que es fundamental resguardar la información sensible mediante la

implementación de ciertas estrategias como reglas de enrutamiento, monitoreo de solicitudes, auditorías, etc.

- Otra sugerencia señala la necesidad de poder importar diagramas ya existentes. Lo cual permitiría continuar con iteraciones de diagramas anteriores generados por la misma herramienta, o incluso con diagramas generados con otras herramientas, siempre y cuando el formato sea compatible.

5.3 Análisis de resultados

El análisis de los resultados obtenidos a partir de las revisiones del prototipo con los arquitectos y las encuestas realizadas demuestra que la herramienta evaluada cumple con las expectativas iniciales en términos de usabilidad, eficiencia y utilidad en la generación de diagramas de componentes. La interfaz intuitiva y la rapidez en la creación de diagramas fueron aspectos destacados por los usuarios, lo que sugiere que la herramienta facilita y optimiza significativamente su flujo de trabajo en el diseño arquitectónico de diagramas de componentes.

A pesar de algunos comentarios respecto a la precisión de los diagramas, los usuarios reconocieron que estos pueden ser mejorados a través de iteraciones adicionales, lo que abre la puerta a futuras optimizaciones del sistema. La incorporación de un LLM se percibió como un valor agregado significativo, brindando asistencia relevante y contextualizada en el proceso de diseño.

5.3.1 Respuesta a la pregunta de investigación

Para responder la pregunta de investigación, respecto al grado de aceptación de la herramienta por parte de los usuarios, hay que enfocarse en las respuestas de la encuesta, particularmente en las preguntas que evalúan aspectos clave relacionados con la

aceptación del prototipo, tales como usabilidad, facilidad, precisión, rapidez, utilidad, satisfacción, y eficiencia percibida. Estas preguntas son la 1, 2, 4, 5, 6, 7 y 8. Basándose en la escala del 1 al 5 utilizada en la encuesta, se considera el total de puntos recibidos en la respuesta de la pregunta, el máximo de puntos posibles y se aplicará la siguiente fórmula:

$$porcentaje = \frac{\text{Puntos obtenidos}}{\text{Puntos posibles}}\ X\ 100$$

- Pregunta 1: (24/30) X 100 = 80.0
- Pregunta 2: (27/30) X 100 = 90.0
- Pregunta 4: (22/30) X 100 = 73.3
- Pregunta 5: (25/30) X 100 = 83.3
- Pregunta 6: (27/30) X 100 = 90.0
- Pregunta 7: (26/30) X 100 = 86.6
- Pregunta 8: Dado que las respuestas fueron un rotundo “SI”, se asume 100%

Finalmente se obtiene un promedio de los porcentajes por respuesta. Eso da un total de 86.17%, el cual sería el porcentaje de aceptación de la herramienta por parte de los usuarios y es la respuesta a la pregunta de investigación.

5.4 Retos y limitaciones

Durante el proceso de validación del prototipo, se han identificado varios retos y limitaciones que impactan tanto la funcionalidad actual como las posibles futuras aplicaciones de la herramienta. Estos desafíos pueden dividirse en dos categorías principales: las limitaciones inherentes a los

Modelos de Lenguaje Grande (LLM) y las limitaciones específicas del prototipo en su implementación actual.

5.4.1 Limitaciones Semánticas de los LLM

Aunque los LLM han demostrado ser herramientas poderosas para la interpretación del lenguaje natural y la generación de contenido basado en texto, presentan varias limitaciones semánticas cuando se trata de tareas altamente estructuradas, como la generación de diagramas de clases. Uno de los principales problemas es la capacidad de los LLM para comprender y mantener relaciones complejas entre entidades dentro de un diagrama, especialmente cuando las reglas y dependencias inherentes no están explícitamente definidas en el input textual.

La generación de diagramas de clases, por ejemplo, requiere que el LLM entienda de manera precisa las relaciones entre clases, interfaces, herencias y asociaciones, lo cual va más allá de simplemente generar una representación gráfica basada en texto. Las limitaciones semánticas actuales dificultan la interpretación correcta de los componentes más abstractos y las relaciones jerárquicas, lo que puede dar lugar a diagramas incompletos o incorrectos. A pesar de que el prototipo ha mostrado buenos resultados en la generación de diagramas de componentes, este desafío sigue siendo una barrera para extender su uso a otros tipos de diagramas más complejos, como los de clases o secuencias.

5.4.2 Limitaciones del Prototipo

El prototipo en su forma actual presenta una serie de limitaciones técnicas y funcionales que han sido identificadas durante las pruebas con los arquitectos.

- Una de las principales es la falta de capacidad de la herramienta para generar distintos tipos de diagramas UML, más allá de los diagramas de componentes. Los arquitectos señalaron que sería valioso contar con opciones para generar diagramas de clases, secuencias y otros tipos UML, lo cual no está contemplado dentro del alcance del proyecto.
- Otra limitación importante es la integración del prototipo con los sistemas corporativos de almacenamiento y gestión de proyectos. El prototipo de la herramienta no permite la comunicación bidireccional con repositorios corporativos, lo que ralentizaría el proceso de replicación de los diseños en entornos colaborativos. Esta limitación afectaría directamente la eficiencia con la que los arquitectos pueden iterar sobre los diagramas generados.
- Un reto adicional mencionado por los arquitectos es la necesidad de asegurar las comunicaciones en caso de utilizar un proveedor de LLM externo en lugar de un modelo local. Dado que el prototipo contempla la posibilidad de cambiar entre distintos proveedores de LLM en tiempo de ejecución, es fundamental implementar estrategias robustas de seguridad que protejan la información sensible manejada durante la generación de diagramas, estrategias tales como reglas de enrutamiento, monitoreo de solicitudes, etc.
- El hecho de que el prototipo solo permita generar diagramas a partir de entradas textuales y no de diagramas previos limita su potencial para iterar sobre diseños ya existentes. Aunque esta funcionalidad no está incluida en la versión actual del prototipo, se sugirió que poder importar diagramas generados

previamente o con otras herramientas permitiría una mejor optimización y adaptación a diferentes flujos de trabajo.

5.4.3 Conclusión

A pesar de los avances logrados con el prototipo, los retos y limitaciones presentados deben ser abordados para asegurar su adopción generalizada y mejorar su funcionalidad. Las limitaciones semánticas de los LLM, en particular, representan un área de investigación clave para mejorar la precisión y utilidad de las herramientas de generación de diagramas. Por otro lado, los desafíos técnicos del prototipo, como la ampliación de su capacidad para generar diversos tipos de diagramas y su integración con sistemas corporativos, ofrecerán nuevas oportunidades de mejora en futuras versiones.

5.5 Propuestas de mejora

Durante el proceso de validación del prototipo, los arquitectos que participaron en las pruebas no solo destacaron las funcionalidades más valiosas de la herramienta, sino que también aportaron sugerencias clave para mejorarla en futuras versiones. Estas propuestas de mejora abordan aspectos tanto técnicos como funcionales, orientados a optimizar la experiencia de usuario y aumentar la versatilidad del prototipo. A continuación, se listan cada una de ellas:

5.5.1 Generación de Diferentes Tipos de Diagramas UML

Una de las propuestas más obvias y esperadas fue la posibilidad de extender las capacidades del prototipo para generar no solo diagramas de componentes, sino también otros tipos de diagramas UML, como diagramas de clases, de secuencia o de actividad. Este avance permitiría a los arquitectos cubrir una gama más amplia de necesidades dentro del ciclo de diseño y

planificación arquitectónica. Aunque esta funcionalidad no está contemplada en el alcance actual del prototipo, la implementación de esta capacidad sería un paso importante para aumentar su aplicabilidad en diferentes fases de desarrollo.

5.5.2 Gestión de Espacios de Trabajo y Proyectos

Los usuarios también señalaron la importancia de contar con un módulo de gestión de proyectos que permita organizar los diagramas generados dentro de espacios de trabajo específicos. Actualmente, el prototipo solicita ingresar el nombre del diagrama en cada iteración, lo que resulta tedioso en proyectos con múltiples versiones o cuando es necesario ajustar el diagrama inicial. Implementar espacios de trabajo donde los usuarios puedan organizar y gestionar sus proyectos y sus respectivas iteraciones facilitaría el flujo de trabajo y mejoraría la eficiencia en la gestión de los diagramas generados.

5.5.3 Mejoras en la Relación de Aspecto de los Diagramas

Otra propuesta de mejora está relacionada con la visualización de los diagramas. Se sugirió que el prototipo debería mejorar la relación de aspecto en la presentación de las imágenes generadas, para evitar distorsiones cuando las dimensiones del visualizador no coinciden con las proporciones originales del diagrama. Asegurar que los diagramas mantengan su integridad visual independientemente del formato de visualización o exportación es fundamental para garantizar su claridad y comprensión.

5.5.4 Exportación de Diagramas en Formatos Adicionales

Actualmente, el prototipo permite la descarga de diagramas en formato .png, sin embargo, algunos usuarios expresaron la necesidad de poder exportar los diagramas en otros formatos, como pdf. Esta funcionalidad mejoraría la compatibilidad de los

diagramas con diversas plataformas de trabajo, además de facilitar su incorporación en reportes técnicos, documentación o presentaciones sin necesidad de realizar conversiones externas.

5.5.5 Integración con Repositorios Corporativos

Otra mejora sugerida por los arquitectos fue la integración del prototipo con repositorios corporativos. Esta funcionalidad permitiría una comunicación bidireccional entre la herramienta y los sistemas de gestión de proyectos empresariales, facilitando la replicación, almacenamiento y actualización de los diagramas en entornos colaborativos. Además, esta integración agilizaría el proceso de diseño, permitiendo que los arquitectos trabajen directamente con la infraestructura tecnológica de su organización, optimizando así el flujo de trabajo y la coherencia en los proyectos.

5.5.6 Seguridad en el Consumo de Modelos de Lenguaje Grandes Externos

Dado que el prototipo permite la opción de utilizar un proveedor de LLM externo en lugar de un modelo local, varios usuarios hicieron hincapié en la importancia de implementar medidas de seguridad robustas para proteger la información sensible. Se sugirió la implementación de estrategias como reglas de enrutamiento, auditoría de solicitudes y monitoreo de tráfico, especialmente en los casos donde se realicen comunicaciones con un LLM externo. Esto aseguraría la integridad de los datos manejados y garantizaría la confidencialidad en los procesos de diseño.

5.5.7 Importación de Diagramas Existentes

Por último, se destacó la necesidad de que el prototipo pueda importar diagramas ya existentes. Esta funcionalidad permitiría a los usuarios continuar con iteraciones sobre diagramas

previamente generados, ya sea por el mismo prototipo o por otras herramientas, siempre que el formato sea compatible. La capacidad de importar y modificar diagramas antiguos no solo incrementaría la flexibilidad de la herramienta, sino que también mejoraría su capacidad para adaptarse a los flujos de trabajo actuales sin necesidad de partir siempre desde cero.

CONCLUSIONES Y RECOMENDACIONES

CONCLUSIONES

1 Durante el desarrollo de este proyecto, se ha desarrollado un prototipo orientado a la generación automática de diagramas de componentes, integrando un Gran Modelo de Lenguaje (LLM) enriquecido con Recuperación Aumentada por Generación (RAG). Este enfoque ha demostrado ser una solución innovadora para abordar los retos tradicionales que enfrentan los arquitectos de software en el diseño manual de diagramas.

2 Los resultados de la validación del prototipo muestran una alta aceptación por parte de los arquitectos de software involucrados en las pruebas, quienes valoraron especialmente la usabilidad y la capacidad de automatización de la herramienta. La reducción significativa en el tiempo necesario para generar diagramas de componentes, pasando de horas a minutos, es un logro destacado que subraya la eficiencia del sistema propuesto. Además, el uso de RAG ha mejorado la precisión contextual de los diagramas, adaptándolos a la realidad de la organización.

3 Aunque el prototipo ha cumplido su objetivo en el ámbito de los diagramas de componentes, su extensión a otros tipos de diagramas UML requerirá investigación y mejoras futuras.

4 En conclusión, este proyecto ha demostrado la viabilidad de utilizar tecnologías avanzadas de IA en el campo del diseño arquitectónico de software. Los resultados obtenidos indican que el prototipo tiene un gran potencial para mejorar la eficiencia en la creación de diagramas, pero también existen áreas clave que deberán abordarse en futuras fases para maximizar su aplicabilidad y robustez.

RECOMENDACIONES

1 Se recomienda establecer un proyecto formal, con sus respectivas fases, para llevar a cabo la implementación de las sugerencias más fácilmente aplicables a corto y mediano plazo, tal como las mejoras en la relación de aspecto de los diagramas, la exportación en múltiples formatos, la integración con sistemas de versionamiento corporativo y la implementación de workspaces. Estas mejoras incrementarán la usabilidad del prototipo y asegurarán que se integre sin problemas en el flujo de trabajo existente de la división de arquitectura.

2 Es esencial garantizar que la herramienta sea lo suficientemente eficiente para su uso diario. Esto implica priorizar mejoras técnicas como la optimización del rendimiento, reduciendo los tiempos de respuesta y aumentando la precisión en la generación de diagramas. El refinamiento continuo de las interfaces y la simplificación del proceso de iteración de diagramas contribuirá a una adopción más fluida por parte de los arquitectos.

    a) Como parte de la estrategia a corto plazo, se recomienda que el prototipo sea utilizado en proyectos piloto reales dentro de la empresa, donde se puedan identificar rápidamente beneficios prácticos y áreas adicionales de mejora. Estos pilotos permitirán medir con mayor precisión la eficacia de la herramienta y su impacto en la productividad, facilitando la toma de decisiones sobre su escalabilidad futura.

    b) Dado el éxito inicial en la generación de diagramas de componentes, se recomienda investigar la viabilidad de aplicar técnicas similares para automatizar la creación de otros tipos de diagramas UML, como diagramas de secuencias y de clases, lo que ampliaría considerablemente el alcance de la herramienta en los procesos de diseño arquitectónico.

BIBLIOGRAFÍA

[1] A. Tariq, M. J. Awan, J. Alshudukhi, T. M. Alam, K. T. Alhamazani, y Z. Meraf, “Software Measurement by Using Artificial Intelligence”, *J. Nanomater.*, vol. 2022, núm. 1, p. 7283171, 2022, doi: 10.1155/2022/7283171.

[2] B. Kim *et al.*, “The Breakthrough Memory Solutions for Improved Performance on LLM Inference”, *IEEE Micro*, vol. 44, núm. 3, pp. 40–48, may 2024, doi: 10.1109/MM.2024.3375352.

[3] Y. Chang *et al.*, “A Survey on Evaluation of Large Language Models”, *ACM Trans Intell Syst Technol*, vol. 15, núm. 3, p. 39:1-39:45, mar. 2024, doi: 10.1145/3641289.

[4] H. Naveed *et al.*, “A Comprehensive Overview of Large Language Models”, el 9 de abril de 2024, *arXiv*: arXiv:2307.06435. doi: 10.48550/arXiv.2307.06435.

[5] E. Y. Zhang, A. D. Cheok, Z. Pan, J. Cai, y Y. Yan, “From Turing to Transformers: A Comprehensive Review and Tutorial on the Evolution and Applications of Generative Transformer Models”, *Sci*, vol. 5, núm. 4, Art. núm. 4, dic. 2023, doi: 10.3390/sci5040046.

[6] J. Rumbaugh, I. Jacobson, y G. Booch, *The unified modeling language reference manual: the definitive reference to the UML from the original designers*, 5. print. en The Addison-Wesley object technology series. Reading, Mass.: Addison-Wesley, 2003.

[7] X. Hou *et al.*, “Large Language Models for Software Engineering: A Systematic Literature Review”, el 10 de abril de 2024, *arXiv*: arXiv:2308.10620. doi: 10.48550/arXiv.2308.10620.

[8] D. De Bari, “Evaluating Large Language Models in Software Design: A Comparative Analysis of UML Class Diagram Generation”, laurea, Politecnico di Torino, 2024. Consultado: el 15 de mayo de 2024. [En línea]. Disponible en: https://webthesis.biblio.polito.it/31177/

[9] R. Lakatos, P. Pollner, A. Hajdu, y T. Joo, “Investigating the performance of Retrieval-Augmented Generation and fine-tuning for the development of AI-driven knowledge-based systems”, el 12 de marzo de 2024, *arXiv*: arXiv:2403.09727. doi: 10.48550/arXiv.2403.09727.

[10] S. Barnett, Z. Brannelly, S. Kurniawan, y S. Wong, “Fine-Tuning or Fine-Failing? Debunking Performance Myths in Large Language Models”, el 17 de junio de 2024, *arXiv*: arXiv:2406.11201. doi: 10.48550/arXiv.2406.11201.

[11] “The C4 model for visualising software architecture”. Consultado: el 2 de septiembre de 2024. [En línea]. Disponible en: https://c4model.com/

[12] “Free UML, BPMN and Agile Tutorials - Learn Step-by-Step”. Consultado: el 6 de septiembre de 2024. [En línea]. Disponible en: https://www.visual-paradigm.com/tutorials/

[13] “Funcionalidades Principales de PowerDesigner”. Consultado: el 6 de septiembre de 2024. [En línea]. Disponible en: https://www.powerdesigner.biz/ES/powerdesigner/powerdesigner-features.html

[14] “Diagramming, Data Visualization and Real-Time Collaboration | Lucidchart”. Consultado: el 6 de septiembre de 2024. [En línea]. Disponible en: https://www.lucidchart.com/pages/product

[15] “Draw Diagrams Online | Gliffy”. Consultado: el 6 de septiembre de 2024. [En línea]. Disponible en: https://www.gliffy.com/products/gliffy-online

[16] “StarUML”. Consultado: el 6 de septiembre de 2024. [En línea]. Disponible en: https://staruml.io/

[17] “Quick Start Guide · ModelioOpenSource/Modelio Wiki · GitHub”. Consultado: el 6 de septiembre de 2024. [En línea]. Disponible en: https://github.com/ModelioOpenSource/Modelio/wiki/Quick-Start-Guide

[18] “USE: UML-based Specification Environment”, SourceForge. Consultado: el 6 de septiembre de 2024. [En línea]. Disponible en: https://sourceforge.net/projects/useocl/

[19] “herramienta de código abierto que utiliza descripciones textuales simples para dibujar hermosos diagramas UML.” Consultado: el 6 de septiembre de 2024. [En línea]. Disponible en: https://plantuml.com/es/

[20] “Class diagrams | Mermaid”. Consultado: el 6 de septiembre de 2024. [En línea]. Disponible en: https://mermaid.js.org/syntax/classDiagram.html

[21] M. Richards y N. Ford, *Fundamentals of Software Architecture: An Engineering Approach*. O’Reilly Media, Inc., 2020.

[22] G. Márquez, H. Astudillo, y R. Kazman, “Architectural tactics in software architecture: A systematic mapping study”, *J. Syst. Softw.*, vol. 197, p. 111558, mar. 2023, doi: 10.1016/j.jss.2022.111558.

[23] E. Gamma, Ed., *Design patterns: elements of reusable object-oriented software*, 39. printing. en Addison-Wesley professional computing series. Boston, Mass. Munich: Addison-Wesley, 2011.

[24] G. Suryanarayana, G. Samarthyam, y T. Sharma, Eds., “Appendix A - Software Design Principles”, en *Refactoring for Software Design Smells*, Boston: Morgan Kaufmann, 2015, pp. 213–215. doi: 10.1016/B978-0-12-801397-7.15001-5.

[25] R. C. Martin, J. Grenning, S. Brown, y K. Henney, *Clean Architecture: a craftsman’s guide to software structure and design*. en Robert C. Martin series. Boston Columbus Indianapolis New York San Francisco Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo: Prentice Hall, 2018.

[26] L. Mehra, *Software Design Patterns for Java Developers: Expert-led Approaches to Build Re-usable Software and Enterprise Applications (English Edition)*. BPB Publications, 2021.

[27] “Software Architecture Patterns, 2nd Edition[Book]”. Consultado: el 10 de septiembre de 2024. [En línea]. Disponible en:

https://www.oreilly.com/library/view/software-architecture-patterns/9781098134280/

[28] A. Cockburn, “Hexagonal architecture”, Alistair Cockburn. Consultado: el 10 de septiembre de 2024. [En línea]. Disponible en: https://alistair.cockburn.us/hexagonal-architecture/

[29] D. Garlan, “Software Architecture”, *Wiley Encycl. Comput. Sci. Eng.*, ene. 2007, Consultado: el 10 de septiembre de 2024. [En línea]. Disponible en: https://www.academia.edu/99223468/Software_Architecture

[30] A. Bellemare, *Building event-driven microservices: leveraging organizational data at scale*, First edition. Sebastopol, CA: O’Reilly Media, 2020.

[31] S. Newman, “Monolith to Microservices”.

[32] M. Jovanović, “What Is a Modular Monolith?” Consultado: el 10 de septiembre de 2024. [En línea]. Disponible en: https://www.milanjovanovic.tech/blog/what-is-a-modular-monolith

[33] M. Ozkaya, “Microservices Killer: Modular Monolithic Architecture”, Design Microservices Architecture with Patterns & Principles. Consultado: el 10 de septiembre de 2024. [En línea]. Disponible en: https://medium.com/design-microservices-architecture-with-patterns/microservices-killer-modular-monolithic-architecture-ac83814f6862

[34] “Reading ‘The C4 model for visualising software architecture’”, Leanpub. Consultado: el 10 de septiembre de 2024. [En línea]. Disponible en: https://leanpub.com/visualising-software-architecture/read_sample

[35] I. Sommerville, “Ingenieria de Software”.

[36] R. S. Pressman, “Ingenieria del Software. Un Enfoque Practico”.

[37] J. Arnowitz, M. Arent, y N. Berger, *Effective Prototyping for Software Makers (Interactive Technologies)*. 2006.

[38] D. Benyon, *Designing interactive systems: a comprehensive guide to HCI, UX and interaction design*, 3. ed. Harlow; Munich: Pearson Education, 2014.

[39] J. Bloch, *Effective Java*, Third edition. Boston Columbus Indianapolis New York San Francisco Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo: Addison-Wesley, 2018.

[40] “Spring Boot”, Spring Boot. Consultado: el 11 de septiembre de 2024. [En línea]. Disponible en: https://spring.io/projects/spring-boot

[41] “IntelliJ IDEA – the Leading Java and Kotlin IDE”, JetBrains. Consultado: el 11 de septiembre de 2024. [En línea]. Disponible en: https://www.jetbrains.com/idea/

[42] “Angular”. Consultado: el 11 de septiembre de 2024. [En línea]. Disponible en: https://angular.dev/

[43] “Visual Studio Code - Code Editing. Redefined”. Consultado: el 11 de septiembre de 2024. [En línea]. Disponible en: https://code.visualstudio.com/

[44] “Welcome to Python.org”, Python.org. Consultado: el 11 de septiembre de 2024. [En línea]. Disponible en: https://www.python.org/doc/

[45] “Download PyCharm: The Python IDE for data science and web development by JetBrains”, JetBrains. Consultado: el 11 de septiembre de 2024. [En línea]. Disponible en: https://www.jetbrains.com/pycharm/download/

[46] “Milvus vector database documentation”. Consultado: el 11 de septiembre de 2024. [En línea]. Disponible en: https://milvus.io/docs

[47] T. B. Brown *et al.*, “Language Models are Few-Shot Learners”, el 22 de julio de 2020, *arXiv*: arXiv:2005.14165. doi: 10.48550/arXiv.2005.14165.

[48] A. Vaswani *et al.*, “Attention Is All You Need”, el 1 de agosto de 2023, *arXiv*: arXiv:1706.03762. doi: 10.48550/arXiv.1706.03762.

[49] “GPT-4”. Consultado: el 11 de septiembre de 2024. [En línea]. Disponible en: https://openai.com/index/gpt-4-research/

[50] “Hugging Face – The AI community building the future.” Consultado: el 11 de septiembre de 2024. [En línea]. Disponible en: https://huggingface.co/

[51] M. Chen *et al.*, “Evaluating Large Language Models Trained on Code”, arXiv.org. Consultado: el 11 de septiembre de 2024. [En línea]. Disponible en: https://arxiv.org/abs/2107.03374v2

[52] “LLM Fine-tuning Use Case: Generate Code Documentation”. Consultado: el 11 de septiembre de 2024. [En línea]. Disponible en: https://predibase.com/documentation-generation

[53] “GitHub Copilot · Your AI pair programmer”, GitHub. Consultado: el 11 de septiembre de 2024. [En línea]. Disponible en: https://github.com/features/copilot

[54] “Eraser – Docs and Diagrams for Engineering Teams”. Consultado: el 11 de septiembre de 2024. [En línea]. Disponible en: https://www.eraser.io/

[55] J. Li, Y. Yuan, y Z. Zhang, “Enhancing LLM Factual Accuracy with RAG to Counter Hallucinations: A Case Study on Domain-Specific Queries in Private Knowledge-Bases”, el 15 de marzo de 2024, *arXiv*: arXiv:2403.10446. doi: 10.48550/arXiv.2403.10446.

[56] M. R. J, K. VM, H. Warrier, y Y. Gupta, “Fine Tuning LLM for Enterprise: Practical Guidelines and Recommendations”, el 23 de marzo de 2024, *arXiv*: arXiv:2404.10779. doi: 10.48550/arXiv.2404.10779.

[57] A. Balaguer *et al.*, “RAG vs Fine-tuning: Pipelines, Tradeoffs, and a Case Study on Agriculture”, el 30 de enero de 2024, *arXiv*: arXiv:2401.08406. doi: 10.48550/arXiv.2401.08406.

[58] S. Alghisi, M. Rizzoli, G. Roccabruna, S. M. Mousavi, y G. Riccardi, “Should We Fine-Tune or RAG? Evaluating Different Techniques to Adapt LLMs for Dialogue”, el 10 de junio de 2024, *arXiv*: arXiv:2406.06399. doi: 10.48550/arXiv.2406.06399.

[59] A. M. Alashqar, “Automatic Generation of Uml Diagrams from Scenario-Based User Requirements”, *Jordanian J. Comput. Inf. Technol.*, vol. 7, núm. 2, 2021, Consultado: el 6 de septiembre de 2024. [En línea]. Disponible en: https://www.proquest.com/docview/2672361426/abstract/2FE1A673EFC44E2EPQ/1

[60] A. Conrardy y J. Cabot, “From Image to UML: First Results of Image Based UML Diagram Generation Using LLMs”, el 17 de abril de 2024, *arXiv*: arXiv:2404.11376. doi: 10.48550/arXiv.2404.11376.

[61] E. A. Abdelnabi, A. M. Maatuk, y M. Hagal, “Generating UML Class Diagram from Natural Language Requirements: A Survey of Approaches and Techniques”, en *2021 IEEE 1st International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering MI-STA*, may 2021, pp. 288–293. doi: 10.1109/MI-STA52233.2021.9464433.

[62] B. Wang, C. Wang, P. Liang, B. Li, y C. Zeng, “How LLMs Aid in UML Modeling: An Exploratory Study with Novice Analysts”, el 26 de abril de 2024, *arXiv*: arXiv:2404.17739. doi: 10.48550/arXiv.2404.17739.

ANEXOS

Anexo 1: Formato de encuesta inicial.

Evaluación de propuesta para herramienta asistente de diseño de software

Encuesta para determinar la necesidad de una herramienta para la generación automática de propuestas base, de diagramas de componentes UML para una solución de
software, a partir del ingreso de algunos requisitos.

* Indica que la pregunta es obligatoria

1. Correo electrónico *

2. ¿Cuál es el nivel de automatización actual en el subproceso que corresponde al diseño del diagrama de componentes para una solución de software? *

Marca solo un óvalo.

	1	2	3	4	5	
Nad	◯	◯	◯	◯	◯	Totalmente automatizado

3. Con la metodología actual, ¿que tan rápido resulta validar la existencia en el sistema, de una funcionalidad (componente) necesaria para un diseño? *

Marca solo un óvalo.

	1	2	3	4	5	
Muy	◯	◯	◯	◯	◯	Muy rápido

4. Con la metodología actual, ¿cuantas horas aproximadamente le toma diseñar un diagrama de componentes para una nueva solución con arquitectura completa? *

Marca solo un óvalo.

- Menor o igual a 2 horas
- Menor o igual a 4 horas
- Menor o igual a 8 horas
- Menor o igual a 16 horas
- Menor o igual a 24 horas
- Menor o igual a 32 horas
- Menor o igual a 40 horas
- Mas de 40 horas

5. Con la metodología actual, una vez diseñado el diagra de componentes, ¿qué tan rápido resulta ajustarlo en caso de requerir cambios? *

Marca solo un óvalo.

	1	2	3	4	5	
Muy	◯	◯	◯	◯	◯	Muy rápido

6. Con la metodología actual, ¿que tan fácil resulta versionar los diseños de las soluciones? *

Marca solo un óvalo.

	1	2	3	4	5	
Muy	◯	◯	◯	◯	◯	Muy fácil

7. ¿Cree usted que le resultaría útil contar con una herramienta que le proporcione de forma automática, un diagrama de componentes base, en formato plantUml, mediante el ingreso de los requisitos de la solución? *

Marca solo un óvalo.

	1	2	3	4	5	
Poc	◯	◯	◯	◯	◯	Muy útil

Anexo 2: Diccionario para estructura de contexto

```
{
  "nombre": "nombre del componente",
  "tipo": "Composite / Core / DB /  etc",
  "capa": "Api Layer / FrontEnd Layer / etc",
  "Regla: clave": "valor",
  "Regla: clave": "valor",
  "funcionalidad": "la funcionalidad del componente"
}
```

Anexo 3: Contrato para generación de contexto

```
{
    "prefix": "component_dictionary",
    "lengSupportedText": 1000,
    "strings": [
        {
            "toEmbed": "La funcionalidad del componente, que forma parte del diccionario",
            "text": "contenido completo del diccionario en formato string"
        }
    ]
}
```

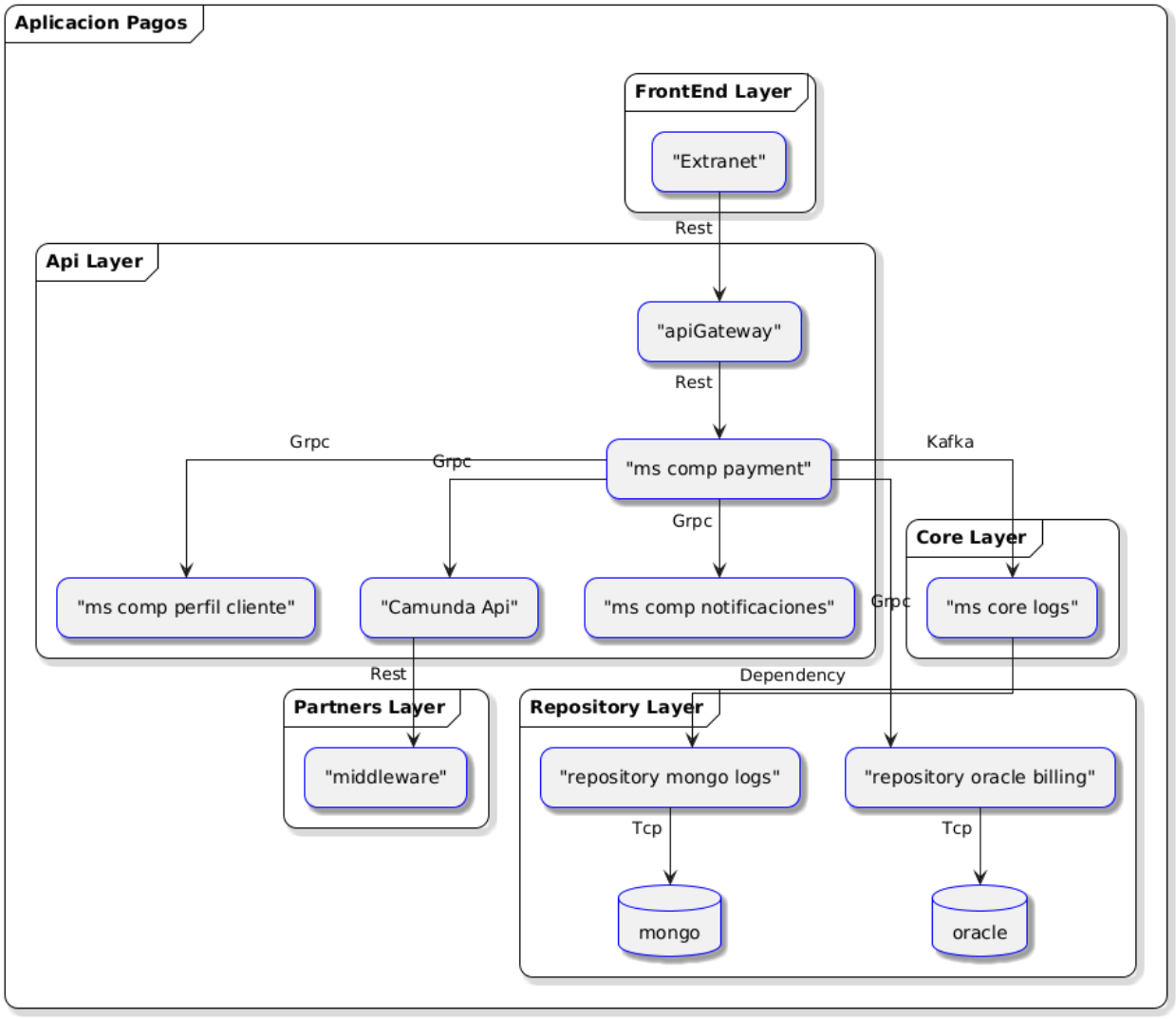
Anexo 4: Diagrama en formato PlantUml

```
@startuml

top to bottom direction
skinparam linetype ortho
skinparam shadowing true
skinparam componentStyle rectangle
skinparam backgroundColor transparent
skinparam frame{
  BorderColor black
  BorderThickness "1"
}
skinparam {
  padding 5
  roundcorner "20"
}
skinparam component {
  BorderThickness 1
  BorderColor Blue
  BorderColor<<nuevo>> Red
}
skinparam database {
  BorderThickness 1
  BorderColor Blue
  BorderColor<<nuevo>> Red
}

hide stereotype
```

```
frame "Aplicacion Pagos" {
  frame "FrontEnd Layer" {
    ["Extranet"] as 1
  }
  frame "Api Layer" {
    ["apiGateway"] as 2
    ["ms comp payment"] as 3
    ["ms comp perfil cliente"] as 4
    ["Camunda Api"] as 5
    ["ms comp notificaciones"] as 6

  frame "Partners Layer" {
    ["middleware"] as 7
  }
  frame "Core Layer" {
    ["ms core logs"] as 8
  }
  frame "Repository Layer" {
    ["repository oracle billing"] as 9
    database "oracle" as 10
    ["repository mongo logs"] as 11
    database "mongo" as 12
  }
  1 --> 2: Rest
  2 --> 3: Rest
  3 --> 5: Grpc
  3 --> 6: Grpc
  3 --> 8: Kafka
  3 --> 9: Grpc
```

```
  5 --> 7 : Rest
  8 --> 11: Dependency
  9 --> 10: Tcp
  11 --> 12: Tcp
  3 --> 4: Grpc
}
@enduml
```

Anexo 5: Diagrama en formato imagen



Anexo 6: Formato de encuesta final

Evaluación de prototipo para herramienta asistente de diseño de software

Encuesta para determinar la aceptación por parte del equipo de arquitectura, para el prototipo de una herramienta que genera de forma automática, diagramas base de componentes de software con ayuda de un LLM y con base en los requisitos de la solución.

* Indica que la pregunta es obligatoria

1. Correo electrónico *

Usabilidad y Experiencia de Usuario

2. ¿Qué tan intuitiva te pareció la interfaz del prototipo? *

Marca solo un óvalo.

	1	2	3	4	5	
Nad	◯	◯	◯	◯	◯	Muy intuitiva

3. ¿El prototipo facilitó la generación de diagramas de componentes en comparación con tus métodos habituales? *

Marca solo un óvalo.

	1	2	3	4	5	
Nad	◯	◯	◯	◯	◯	Mucho

4. ¿Encontraste algún obstáculo técnico al utilizar el prototipo? Si es así, por favor indícalo. *

_____

Funcionalidad y Rendimiento

5. ¿Cómo calificarías la precisión de los diagramas generados por el prototipo? *

Marca solo un óvalo.

	1	2	3	4	5	
Nad	◯	◯	◯	◯	◯	Muy preciso

6. ¿Qué tan rápido es el proceso de generación de diagramas en el prototipo? *

Marca solo un óvalo.

	1	2	3	4	5	
Nad	◯	◯	◯	◯	◯	Muy rápido

7. ¿Te resultó útil la asistencia del LLM en la generación de diagramas? *

Marca solo un óvalo.

	1	2	3	4	5	
Nad	◯	◯	◯	◯	◯	Muy útil

Percepción general

8. ¿Cómo evalúas tu satisfacción general con el prototipo? *

Marca solo un óvalo.

	1	2	3	4	5	
Nad	◯	◯	◯	◯	◯	Muy satisfecho

9. ¿Consideras que el prototipo podría aumentar la eficiencia en tus proyectos? *

Marca solo un óvalo.

- ◯ Si
- ◯ No

10. ¿Qué aspectos del prototipo te parecieron más valiosos? *

11. ¿Qué funcionalidades adicionales te gustaría que el prototipo incorporara para mejorar su integración?. Redáctalo por ítems *