



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Trabajo fin de Grado

Grado en I.I. en Tecnologías Informáticas

**Optimización de la respuesta de los LLMs en el proceso de Ingeniería de
Requisitos**

**Realizado por
Antonio Suero Baeza**

**Dirigido por
Nombre del Tutor**

**Departamento
Lenguajes y Sistemas Informáticos**

Sevilla, Febrero de 2026

Resumen

Los Modelos de Lenguaje Grande (LLMs) han abierto nuevas posibilidades para la automatización de tareas de Ingeniería de Requisitos (IR), pero su aplicación práctica presenta incógnitas relevantes: ¿qué tan capaces son los modelos que pueden ejecutarse de forma local, sin enviar datos a servidores externos? ¿Qué estrategia de prompting produce los mejores resultados? ¿A qué coste computacional?

Una revisión sistemática reciente de 27 estudios sobre LLMs en IR revela que **ningún trabajo previo ha evaluado modelos locales en este dominio**. Este Trabajo de Fin de Grado aborda directamente ese vacío, presentando una evaluación empírica y sistemática de seis modelos LLM—tres locales (Qwen 2.5 7B, Llama 3.1 8B, Llama 3.2 3B, ejecutados mediante Ollama) y tres accedidos mediante API (Llama 3.1 70B, Llama 3.1 8B, Mistral 7B, a través de NVIDIA NIM)—sobre cinco tareas de IR: clasificación funcional/no funcional (A1), detección de ambigüedad (A2), evaluación de completitud (A3), detección de inconsistencias (V1) y evaluación de testabilidad (V2).

Se implementa un diseño factorial completo de 750 configuraciones (5 tareas \times 6 modelos \times 5 estrategias \times 5 iteraciones) con cinco estrategias de prompting: *Question Refinement*, *Cognitive Verifier*, *Persona + Context*, *Few-Shot* y *Chain of Thought*. Los resultados se analizan con ANOVA, pruebas t de Welch y tamaños del efecto (Cohen's d) para responder a tres preguntas de investigación sobre (1) la brecha de rendimiento local-API, (2) la estrategia de prompt óptima, y (3) los trade-offs rendimiento-coste-privacidad.

Como contribuciones adicionales, se desarrollan datasets anotados para las tareas de completitud y testabilidad, que carecen de recursos públicos consolidados, y se construye un sistema completo que incluye el pipeline experimental con soporte a checkpoint/reanudación, una API REST (FastAPI) y un frontend interactivo (Streamlit) para el análisis de documentos de requisitos en tiempo real.

Palabras clave: Ingeniería de Requisitos, Modelos de Lenguaje Grande, prompting, evaluación empírica, modelos locales, NVIDIA NIM, Ollama, clasificación de requisitos.

Abstract

Large Language Models (LLMs) have opened new possibilities for the automation of Requirements Engineering (RE) tasks, yet practical questions remain open: how capable are models that can be run locally, without sending data to external servers? Which prompting strategy produces the best results? At what computational cost?

A recent systematic review of 27 studies on LLMs in RE reveals that **no prior work has evaluated local models in this domain**. This Bachelor's Thesis directly addresses that gap by presenting a systematic empirical evaluation of six LLMs —three local models (Qwen 2.5 7B, Llama 3.1 8B, Llama 3.2 3B, run via Ollama) and three API-accessed models (Llama 3.1 70B, Llama 3.1 8B, Mistral 7B, via NVIDIA NIM)— on five RE tasks: functional/non-functional classification (A1), ambiguity detection (A2), completeness evaluation (A3), inconsistency detection (V1), and testability evaluation (V2).

A full factorial design of 750 configurations is implemented ($5 \text{ tasks} \times 6 \text{ models} \times 5 \text{ strategies} \times 5 \text{ iterations}$) with five prompting strategies: Question Refinement, Cognitive Verifier, Persona + Context, Few-Shot, and Chain of Thought. Results are analysed using ANOVA, Welch's t-tests, and effect sizes (Cohen's d) to answer three research questions on (1) the local-API performance gap, (2) the optimal prompting strategy, and (3) performance-cost-privacy trade-offs.

As additional contributions, annotated datasets are built for the completeness and testability tasks (which lack public benchmarks), and a complete system is developed comprising the experimental pipeline with checkpoint/resume support, a REST API (FastAPI), and an interactive frontend (Streamlit) for real-time requirements document analysis.

Keywords: Requirements Engineering, Large Language Models, prompt engineering, empirical evaluation, local models, NVIDIA NIM, Ollama, requirements classification.

Índice general

Índice general	III
Índice de cuadros	VI
Índice de figuras	VII
1 Introducción	1
1.1 Motivación	1
1.2 Objetivos del trabajo	2
1.3 Preguntas de investigación	2
1.4 Estructura del documento	3
2 Estado del Arte	4
2.1 Ingeniería de Requisitos	4
2.1.1 Tareas principales de la Ingeniería de Requisitos	4
2.1.2 Desafíos actuales en IR	5
2.2 Modelos de Lenguaje Grande (LLMs)	5
2.2.1 Arquitectura Transformer	5
2.2.2 Evolución y modelos actuales	6
2.2.3 Tipología de modelos según el modo de acceso	6
2.2.4 Modelos evaluados en este trabajo	6
2.3 Aplicación de LLMs en Ingeniería de Requisitos	6
2.3.1 Revisión sistemática de Cheng et al.	6
2.3.2 Trabajos relacionados en clasificación de requisitos	7
2.3.3 Detección de ambigüedad y otros atributos de calidad	7
2.4 Estrategias de Prompting	7
2.5 Gap identificado y motivación del estudio	8
3 Metodología	9
3.1 Diseño experimental	9
3.2 Conjuntos de datos	9
3.2.1 A1 — Clasificación Funcional / No Funcional	9
3.2.2 A2 — Detección de Ambigüedad	10
3.2.3 A3 — Evaluación de Completitud	10
3.2.4 V1 — Detección de Inconsistencias	10
3.2.5 V2 — Evaluación de Testabilidad	10
3.3 Modelos evaluados	11
3.3.1 Modelos locales (Ollama)	11
3.3.2 Modelos de API (NVIDIA NIM)	11
3.3.3 Criterios de selección	11

3.4	Estrategias de prompting	12
3.4.1	Question Refinement (QR)	12
3.4.2	Cognitive Verifier (CV)	12
3.4.3	Persona + Context (PC)	12
3.4.4	Few-Shot (FS)	12
3.4.5	Chain of Thought (CoT)	12
3.5	Métricas de evaluación	12
3.5.1	Métricas de rendimiento	12
3.5.2	Tasa de respuestas inválidas (invalid_rate)	13
3.5.3	Rendimiento computacional	13
3.5.4	Análisis estadístico	13
3.6	Procedimiento experimental	13
3.6.1	Pipeline de ejecución	13
3.6.2	Parámetros de generación	14
3.6.3	Gestión de fallos y reproducibilidad	14
3.6.4	Hardware utilizado	14
4	Sistema desarrollado	15
4.1	Arquitectura general	15
4.2	Pipeline de experimentación (experiment.py)	15
4.2.1	Clase ExperimentRunner	15
4.2.2	Mecanismo de checkpoint y reanudación	16
4.2.3	Modo dry-run	16
4.3	Módulo de prompts (prompts.py)	16
4.3.1	Estructura de los prompts	16
4.3.2	Función build_prompt	16
4.3.3	Parsers de respuesta por tarea	17
4.3.4	Prompts combinados y pipeline de documentos	17
4.4	Wrappers de modelos (models.py)	17
4.4.1	Interfaz unificada	17
4.4.2	OllamaModel	17
4.4.3	NVIDIAANIMModel	17
4.4.4	Reintentos con backoff exponencial	18
4.5	Módulo de análisis (analysis.py)	18
4.5.1	Gráficos generados	18
4.5.2	Análisis estadístico	18
4.5.3	Generación de informes consolidados	18
4.6	API REST (api.py)	18
4.6.1	Descripción general	18
4.6.2	Endpoints principales	19
4.6.3	Validación y documentación	19
4.7	Frontend Streamlit (app.py)	19
4.7.1	Descripción general	19
4.7.2	Páginas del frontend	19
4.7.3	Sistema de advertencias	20
5	Resultados y análisis	21
5.1	Resultados por tarea	21
5.1.1	A1 — Clasificación Funcional / No Funcional	21
5.1.2	A2 — Detección de ambigüedad	21
5.1.3	A3 — Evaluación de completitud	22
5.1.4	V1 — Detección de inconsistencias	22

5.1.5	V2 — Evaluación de testabilidad	23
5.2	Comparación local vs. API (RQ1)	23
5.3	Comparación por estrategia de prompting (RQ2)	24
5.4	Análisis de trade-offs (RQ3)	25
5.5	Análisis estadístico	25
6	Conclusiones	27
6.1	Respuesta a las preguntas de investigación	27
6.1.1	RQ1 — ¿En qué medida difieren los modelos locales y los modelos de API en tareas de Ingeniería de Requisitos?	27
6.1.2	RQ2 — ¿Qué estrategia de prompting produce el mejor rendimiento?	27
6.1.3	RQ3 — ¿Cuáles son los trade-offs entre rendimiento, coste y privacidad?	28
6.2	Contribuciones del trabajo	28
6.3	Limitaciones	29
6.4	Trabajo futuro	29
	Referencias	30

Índice de cuadros

3.1	Distribución del dataset A1 (Clasificación F/NF, muestra experimental de 100) . .	10
3.2	Distribución del dataset A2 (Ambigüedad)	10
3.3	Distribución del dataset A3 (Compleitud)	10
3.4	Distribución del dataset V1 (Inconsistencias)	10
3.5	Distribución del dataset V2 (Testabilidad)	11
3.6	Modelos locales evaluados	11
3.7	Modelos de API evaluados (NVIDIA NIM)	11
3.8	Parámetros de generación del modelo	14
3.9	Hardware del entorno de experimentación	14
4.1	Endpoints de la API REST	19
5.1	Mejores resultados en A1 (Clasificación F/NF)	21
5.2	Mejores resultados en A2 (Detección de ambigüedad)	22
5.3	F1 medio por modelo en A3 (Compleitud)	22
5.4	Mejores resultados en V1 (Detección de inconsistencias)	22
5.5	Mejores resultados en V2 (Evaluación de testabilidad)	23
5.6	Comparación local vs. API por tarea (F1 medio)	23
5.7	Ranking global de modelos (F1 medio sobre las 5 tareas)	24
5.8	F1 medio por estrategia de prompting y tarea	24
5.9	Eficiencia media de los modelos (F1/s y latencia)	25
5.10	Resultados ANOVA de un factor por tarea (métrica: F1)	25

Índice de figuras

Introducción

La Ingeniería de Requisitos (IR) constituye una de las fases más críticas del ciclo de vida del software. Errores cometidos en esta etapa se propagan al resto del desarrollo, incrementando de forma exponencial el coste de corrección (Boehm, 1988). La calidad de los requisitos afecta directamente a la correcta implementación del sistema: un requisito ambiguo, incompleto o inconsistente puede dar lugar a malentendidos entre cliente y desarrollador que solo se detectan en fases avanzadas del proyecto.

En paralelo, los Modelos de Lenguaje Grande (LLM, del inglés *Large Language Models*) han experimentado un desarrollo vertiginoso en los últimos años. Modelos como GPT-4, Llama o Mistral demuestran capacidades avanzadas de comprensión del lenguaje natural que los hacen candidatos prometedores para automatizar tareas de análisis de texto en contextos de ingeniería del software.

1.1– Motivación

La revisión sistemática de la literatura realizada por Cheng, Xia, Treude, y Hassan (2024) sobre el uso de LLMs en Ingeniería de Requisitos analiza 27 estudios publicados entre 2020 y 2024. Uno de los hallazgos más relevantes de dicha revisión es que **ninguno de los estudios evaluados incorpora modelos de lenguaje ejecutados de forma local**. Todos los trabajos revisados hacen uso exclusivo de modelos de pago accesibles mediante API, principalmente variantes de la familia GPT de OpenAI.

Esta ausencia implica una brecha significativa en el conocimiento disponible sobre el rendimiento comparado entre:

- Modelos **locales** (ejecutados en hardware propio, sin envío de datos a servidores externos).
- Modelos **en la nube** accedidos mediante API (mayor capacidad, mayor coste, menor privacidad).

Dicha brecha tiene implicaciones prácticas notables. Las organizaciones que trabajan con documentación de requisitos confidencial —como empresas de defensa, sector financiero o administración pública— no pueden utilizar APIs externas por razones de privacidad y soberanía de los datos. Estas organizaciones necesitan conocer si los modelos locales, generalmente de menor tamaño y menor coste de inferencia, son capaces de ofrecer un rendimiento aceptable en tareas de IR.

Por otro lado, Ronanki, Cabrera, y Engström (2024) evaluaron tres patrones de prompt con GPT-3.5 en tareas de IR, concluyendo que la elección de la estrategia de prompting afecta significativamente al rendimiento. Sin embargo, su estudio se limita a un único modelo y no investiga

si dichos patrones generalizan a otros modelos, ya sean de mayor o menor tamaño, locales o en la nube.

Este Trabajo de Fin de Grado aborda directamente las dos brechas identificadas: la ausencia de evaluaciones de modelos locales en IR y la falta de comparativas sistemáticas de estrategias de prompting entre modelos heterogéneos.

1.2– Objetivos del trabajo

El objetivo principal de este trabajo es **evaluar y comparar el rendimiento de modelos LLM locales y de API en cinco tareas de Ingeniería de Requisitos**, bajo cinco estrategias de prompting distintas, con el fin de proporcionar guías basadas en evidencia para su aplicación práctica.

Los objetivos específicos son los siguientes:

1. Implementar un **pipeline de experimentación reproducible** que permita evaluar de forma automática cualquier combinación de modelo, tarea y estrategia de prompting.
2. Evaluar **seis modelos LLM** —tres locales (Qwen 2.5 7B, Llama 3.1 8B, Llama 3.2 3B) y tres accedidos mediante la plataforma NVIDIA NIM (Llama 3.1 70B, Llama 3.1 8B, Mistral 7B)— sobre cinco tareas de IR.
3. Comparar **cinco estrategias de prompting** (*Question Refinement*, *Cognitive Verifier*, *Persona + Context*, *Few-Shot* y *Chain of Thought*) con respecto a su efecto sobre la calidad de las predicciones.
4. Analizar los **trade-offs** entre rendimiento, coste computacional y privacidad de datos, de cara a orientar la toma de decisiones en proyectos reales.
5. Desarrollar un **sistema completo** que integre el pipeline experimental con una API REST y un frontend interactivo, permitiendo el análisis de documentos de requisitos en tiempo real.
6. Contribuir con **datasets anotados** para las tareas de evaluación de completitud y testabilidad, áreas en las que no existen recursos públicos consolidados.

1.3– Preguntas de investigación

Para estructurar el análisis y facilitar la interpretación de los resultados, el trabajo se orienta en torno a tres preguntas de investigación:

- RQ1 — Rendimiento local vs. API:** ¿Existen diferencias estadísticamente significativas entre el rendimiento de los modelos locales y los modelos de API en las tareas de Ingeniería de Requisitos evaluadas?
- RQ2 — Estrategia de prompting óptima:** ¿Qué estrategia de prompting produce el mejor rendimiento para cada combinación de modelo y tarea, y es posible identificar una estrategia dominante de forma general?
- RQ3 — Trade-offs:** ¿Cuáles son los compromisos entre rendimiento (F1-score), coste computacional (tokens por segundo, latencia) y privacidad de los datos cuando se comparan modelos locales y de API?

1.4– Estructura del documento

El presente documento se organiza en los siguientes capítulos:

Capítulo 2 — Estado del Arte: Revisión de los conceptos fundamentales de Ingeniería de Requisitos, descripción de los Modelos de Lenguaje Grande, análisis de los trabajos previos sobre la aplicación de LLMs en IR, y caracterización del gap identificado.

Capítulo 3 — Metodología: Descripción detallada del diseño experimental, los conjuntos de datos utilizados, los modelos evaluados, las estrategias de prompting implementadas, las métricas de evaluación y el procedimiento experimental.

Capítulo 4 — Sistema desarrollado: Presentación de la arquitectura del sistema, incluyendo el pipeline de experimentación, los módulos de prompts y modelos, el motor de análisis estadístico, la API REST y el frontend interactivo.

Capítulo 5 — Resultados y análisis: Exposición de los resultados obtenidos en los experimentos, comparativa por tarea, modelo y estrategia, y análisis estadístico (ANOVA, pruebas t , tamaño del efecto).

Capítulo 6 — Conclusiones: Respuesta a las preguntas de investigación, limitaciones del estudio, implicaciones prácticas y líneas de trabajo futuro.

Estado del Arte

Este capítulo proporciona el marco teórico necesario para contextualizar el trabajo. Se revisan, en primer lugar, los fundamentos de la Ingeniería de Requisitos y las tareas que la componen. A continuación se describen los Modelos de Lenguaje Grande, su arquitectura y tipología. Seguidamente se analiza el estado actual de la aplicación de LLMs en IR, centrándonos en las revisiones sistemáticas más recientes. Por último se caracterizan las principales estrategias de prompting y se delimita el gap que motiva este trabajo.

2.1– Ingeniería de Requisitos

La Ingeniería de Requisitos (IR) es la disciplina de la ingeniería del software que se ocupa de identificar, documentar, analizar, priorizar y validar los requisitos de un sistema software antes de iniciar su construcción (Sommerville, 2016). Su importancia radica en que los errores de requisitos son los más costosos de corregir: estudios clásicos de Boehm (1988) demuestran que corregir un error en fase de mantenimiento puede costar hasta 100 veces más que hacerlo en la fase de requisitos.

Los requisitos de software se clasifican habitualmente en dos grandes categorías (Cleland-Huang, Settini, Zou, y Solc, 2007):

Requisitos Funcionales (RF): Describen *qué* debe hacer el sistema, es decir, las funciones, capacidades y comportamientos que el sistema debe proporcionar. Ejemplo: *“El sistema permitirá al usuario autenticarse mediante usuario y contraseña”*.

Requisitos No Funcionales (RNF): Describen *cómo* debe comportarse el sistema, expresando restricciones de calidad como rendimiento, seguridad, usabilidad o mantenibilidad. Ejemplo: *“El tiempo de respuesta del sistema no superará los 2 segundos en el 95 % de las peticiones”*.

2.1.1. Tareas principales de la Ingeniería de Requisitos

Las tareas que componen el proceso de IR son diversas. Las más relevantes para este trabajo son:

- **Clasificación Funcional / No Funcional (A1):** Determinar la naturaleza de cada requisito. La clasificación errónea puede llevar a que requisitos de rendimiento o seguridad no reciban el tratamiento adecuado.

- **Detección de Ambigüedad (A2):** Identificar requisitos que admiten múltiples interpretaciones. Femmer, Mund, y Fernández (2017) proponen un catálogo de *requirements smells* —patrones lingüísticos asociados a ambigüedad— que incluye términos vagos, referencias implícitas y condiciones incompletas.
- **Evaluación de Completitud (A3):** Comprobar si un requisito proporciona toda la información necesaria para su implementación sin tener que recurrir a suposiciones.
- **Detección de Inconsistencias (V1):** Verificar que dos o más requisitos no se contradicen entre sí, lo que podría originar comportamientos indefinidos en el sistema.
- **Evaluación de Testabilidad (V2):** Determinar si un requisito puede ser verificado mediante una prueba objetiva y reproducible.

2.1.2. Desafíos actuales en IR

A pesar de décadas de investigación, la IR sigue enfrentando retos significativos en la práctica industrial (Cheng y cols., 2024):

- Los documentos de requisitos suelen redactarse en lenguaje natural, lo que introduce ambigüedad intrínseca difícil de eliminar.
- La revisión manual de grandes documentos es costosa en tiempo y propensa a errores humanos.
- La trazabilidad entre requisitos y código es difícil de mantener en proyectos de larga duración.
- No existe una métrica universal de calidad de requisitos aceptada por toda la comunidad.

Estos desafíos motivan la búsqueda de herramientas automatizadas que asistan a los ingenieros de requisitos. El procesamiento del lenguaje natural (PLN) ha proporcionado enfoques relevantes durante décadas, y más recientemente los LLMs han abierto nuevas posibilidades.

2.2– Modelos de Lenguaje Grande (LLMs)

Un Modelo de Lenguaje Grande es un sistema de aprendizaje profundo entrenado sobre vastas cantidades de texto para predecir la distribución de probabilidad del siguiente token dada una secuencia de tokens precedentes. La arquitectura predominante en los LLMs actuales es el Transformador (*Transformer*), introducido por Vaswani y cols. (2017).

2.2.1. Arquitectura Transformer

El Transformador utiliza mecanismos de atención multi-cabeza que permiten al modelo relacionar cualquier par de posiciones de la secuencia de entrada independientemente de su distancia. Esto supera las limitaciones de las redes recurrentes (RNN, LSTM) que procesaban el texto de forma secuencial y sufrían de gradiente desvaneciente en secuencias largas.

La arquitectura *decoder-only*, adoptada por modelos como GPT, Llama y Mistral, genera texto de forma autorregresiva: el modelo predice un token, lo concatena a la entrada y predice el siguiente. Este diseño resulta especialmente adecuado para tareas de generación de texto, incluyendo la clasificación cuando se formula como tarea generativa.

2.2.2. Evolución y modelos actuales

La escala de los LLMs ha crecido de forma exponencial. GPT-3 (Brown y cols., 2020) demostró con 175.000 millones de parámetros capacidades emergentes como el aprendizaje en contexto (*in-context learning*) que no eran predecibles extrapolando modelos más pequeños.

Posteriormente, el ajuste por instrucciones (*instruction tuning*) y el aprendizaje por refuerzo con retroalimentación humana (RLHF, *Reinforcement Learning from Human Feedback*) mejoraron notablemente la alineación de los modelos con las instrucciones del usuario, dando lugar a modelos como ChatGPT, GPT-4, y las familias Llama 3 y Mistral.

2.2.3. Tipología de modelos según el modo de acceso

Desde el punto de vista práctico, los LLMs se pueden clasificar según su modo de acceso:

Modelos locales: Se ejecutan íntegramente en el hardware del usuario, sin enviar datos a servidores externos. Requieren suficiente memoria RAM o VRAM para almacenar los pesos del modelo. La cuantización (*quantization*) permite reducir el tamaño de los pesos (de 16 bits a 4 u 8 bits) a costa de una pequeña pérdida de precisión, haciendo posible ejecutar modelos de 7–8B de parámetros en GPUs de consumo (8 GB VRAM).

Modelos de API: Se acceden mediante llamadas HTTP a servicios en la nube. El proveedor gestiona la infraestructura y el usuario paga por los tokens procesados. Ofrecen acceso a modelos mucho más grandes (70B–405B de parámetros) sin necesidad de hardware especializado, pero implican costes por uso y el envío de datos fuera del control del usuario.

2.2.4. Modelos evaluados en este trabajo

Los modelos seleccionados en este estudio representan un espectro amplio de tamaños y modalidades de acceso:

- **Qwen 2.5 7B Instruct** (local, cuantizado Q5_K_M): Desarrollado por Alibaba Cloud. Destaca por su rendimiento en tareas de razonamiento con un tamaño reducido.
- **Llama 3.1 8B Instruct** (local, cuantizado Q4_K_M): Familia Meta AI, de referencia en modelos open-source.
- **Llama 3.2 3B Instruct** (local, cuantizado Q4_K_M): Variante ultraligera de Meta AI para hardware con recursos muy limitados.
- **Llama 3.1 70B Instruct** (API, NVIDIA NIM): La variante más grande de Llama 3.1, con capacidades comparables a GPT-4 en muchos benchmarks.
- **Llama 3.1 8B Instruct** (API, NVIDIA NIM): Misma arquitectura que la versión local pero servida en infraestructura optimizada.
- **Mistral 7B Instruct v0.3** (API, NVIDIA NIM): Desarrollado por Mistral AI, conocido por su eficiencia y rendimiento en benchmarks estándar.

2.3– Aplicación de LLMs en Ingeniería de Requisitos

2.3.1. Revisión sistemática de Cheng et al.

La revisión sistemática de Cheng y cols. (2024) constituye la referencia más completa y reciente sobre el uso de LLMs en IR. El estudio analiza 27 artículos publicados entre 2017 y 2024, e identifica las siguientes tendencias:

- La tarea más estudiada es la **clasificación de requisitos** (F/NF), presente en 14 de los 27 estudios.
- GPT-3.5 y GPT-4 son los modelos dominantes, apareciendo en el 74 % de los estudios.
- El **0 % de los estudios evalúa modelos de código abierto ejecutados de forma local**, lo que constituye el principal gap identificado.
- Los enfoques de prompting *zero-shot* y *few-shot* son los más empleados, mientras que técnicas más elaboradas como *Chain of Thought* apenas han sido exploradas en el contexto de IR.

2.3.2. Trabajos relacionados en clasificación de requisitos

La clasificación de requisitos F/NF ha recibido atención significativa antes de la era de los LLMs, mediante enfoques basados en máquinas de vectores de soporte (SVM) y aprendizaje profundo sobre el dataset PROMISE (Cleland-Huang y cols., 2007). Con la llegada de los LLMs, Ronanki y cols. (2024) demuestran que GPT-3.5 alcanza un F1-score superior al 90 % en esta tarea utilizando prompts adecuados, superando a los clasificadores tradicionales sin necesidad de entrenamiento específico.

2.3.3. Detección de ambigüedad y otros atributos de calidad

La detección automática de ambigüedad en requisitos es una tarea más desafiante. Femmer y cols. (2017) proponen un enfoque basado en reglas lingüísticas que detecta *smells* de requisitos, mientras que trabajos más recientes exploran el uso de modelos BERT y LLMs para esta tarea. Cheng y cols. (2024) reportan que la detección de ambigüedad y la evaluación de la completitud tienen una representación mucho menor en la literatura, lo que indica oportunidades de investigación.

2.4– Estrategias de Prompting

La ingeniería de prompts (*prompt engineering*) hace referencia al diseño sistemático de las instrucciones que se proporcionan a un LLM para maximizar la calidad de sus respuestas. White y cols. (2024) identifican 26 patrones de prompt reutilizables, análogos a los patrones de diseño en ingeniería del software, aplicados a la mejora de la calidad del código. Ronanki y cols. (2024) adaptan tres de estos patrones al contexto de IR.

Las cinco estrategias evaluadas en este trabajo son:

Question Refinement (QR): El modelo recibe instrucción de reformular mentalmente la pregunta o tarea antes de responder, clarificando el enunciado cuando sea ambiguo. Basado en el patrón homónimo de White y cols. (2024).

Cognitive Verifier (CV): Se solicita al modelo que descomponga el problema en subpreguntas más simples, responda cada una por separado y sintetice una respuesta final. Fomenta el razonamiento estructurado.

Persona + Context (PC): Se asigna al modelo una identidad experta (por ejemplo, “ingeniero de requisitos senior con 10 años de experiencia en IEEE 830”) y se proporciona contexto sobre la tarea. Influye en el *prior* del modelo sobre el dominio.

Few-Shot (FS): Se incluyen varios ejemplos etiquetados en el prompt (entre 3 y 5) antes de presentar el caso a clasificar. Técnica estándar de aprendizaje en contexto demostrada por Brown y cols. (2020).

Chain of Thought (CoT): Se solicita explícitamente al modelo que razone paso a paso antes de emitir su respuesta final, facilitando la introspección del proceso de decisión. Introducido por Wei y cols. (2022).

2.5– Gap identificado y motivación del estudio

La síntesis de la revisión de la literatura permite identificar con claridad las brechas que este trabajo pretende cubrir:

1. **Ausencia de modelos locales:** Como señala Cheng y cols. (2024), ningún estudio previo evalúa modelos LLM ejecutados localmente en tareas de IR. Este trabajo es, hasta donde se tiene conocimiento, el primero en hacerlo de forma sistemática y con múltiples modelos.
2. **Comparativa multi-estrategia y multi-modelo:** Los estudios existentes evalúan, como mucho, uno o dos modelos con un subconjunto reducido de estrategias de prompting. Este trabajo implementa un diseño factorial completo de 6 modelos \times 5 estrategias.
3. **Cobertura de tareas poco estudiadas:** Las tareas de evaluación de completitud y testabilidad carecen de datasets públicos consolidados. Este trabajo contribuye con datasets anotados para ambas tareas.
4. **Análisis de trade-offs:** No existen estudios que analicen explícitamente el compromiso entre calidad de la respuesta y coste computacional (latencia, tokens por segundo) en el contexto de IR, información esencial para la adopción práctica.

Metodología

Este capítulo describe el diseño experimental adoptado, los conjuntos de datos utilizados, los modelos evaluados, las estrategias de prompting implementadas, las métricas de evaluación y el procedimiento experimental seguido.

3.1– Diseño experimental

El estudio adopta un diseño factorial completo para maximizar la comparabilidad entre condiciones experimentales. Los factores del diseño son:

- **Tarea** (5 niveles): clasificación F/NF, detección de ambigüedad, evaluación de completitud, detección de inconsistencias, evaluación de testabilidad.
- **Modelo** (6 niveles): tres locales y tres de API.
- **Estrategia de prompting** (5 niveles): QR, CV, PC, FS, CoT.
- **Iteración** (5 niveles): se ejecuta cada configuración con 5 semillas distintas para estabilizar los resultados ante la aleatoriedad del muestreo del modelo.

El número total de configuraciones evaluadas es:

$$5 \text{ tareas} \times 6 \text{ modelos} \times 5 \text{ estrategias} \times 5 \text{ iteraciones} = 750 \text{ configuraciones}$$

Cada configuración genera una predicción por muestra del dataset de la tarea correspondiente. El total de llamadas al modelo se calcula como el número de configuraciones multiplicado por el tamaño del dataset de cada tarea.

3.2– Conjuntos de datos

3.2.1. A1 — Clasificación Funcional / No Funcional

Se utiliza una versión ampliada del dataset PROMISE NFR, propuesto originalmente por Cleland-Huang y cols. (2007) y empleado como referencia estándar en la literatura de clasificación de requisitos. El conjunto completo contiene 625 requisitos (365 funcionales y 260 no funcionales); para los experimentos se extrae una muestra aleatoria de 100 requisitos manteniendo la distribución proporcional original (aproximadamente 58 % F / 42 % NF).

Etiqueta	Muestras	Porcentaje
Funcional (F)	58	58 %
No Funcional (NF)	42	42 %
Total	100	100 %

Cuadro 3.1: Distribución del dataset A1 (Clasificación F/NF, muestra experimental de 100)

3.2.2. A2 — Detección de Ambigüedad

El dataset de ambigüedad contiene 262 requisitos anotados como ambiguos o no ambiguos. Los requisitos se anotaron manualmente siguiendo los criterios de *requirements smells* definidos por Femmer y cols. (2017): un requisito es ambiguo si admite múltiples interpretaciones válidas o utiliza términos vagos sin cuantificar. La distribución resultante muestra un ligero desbalance favorable a los requisitos ambiguos.

Etiqueta	Muestras	Porcentaje
Ambiguo (1)	139	53 %
No Ambiguo (0)	123	47 %
Total	262	100 %

Cuadro 3.2: Distribución del dataset A2 (Ambigüedad)

3.2.3. A3 — Evaluación de Completitud

El dataset de completitud es una contribución original de este trabajo, dado que no existen conjuntos de datos públicos para esta tarea en el contexto de IR. Contiene 150 requisitos anotados como completos o incompletos, donde un requisito se considera incompleto si omite sujeto, precondition, postcondition o parámetros necesarios para su implementación.

Etiqueta	Muestras	Porcentaje
Completo (1)	75	50 %
Incompleto (0)	75	50 %
Total	150	100 %

Cuadro 3.3: Distribución del dataset A3 (Completitud)

3.2.4. V1 — Detección de Inconsistencias

El dataset de inconsistencias contiene 30 pares de requisitos, cada uno etiquetado como consistente o inconsistente. Dos requisitos son inconsistentes si afirman condiciones mutuamente excluyentes sobre el mismo aspecto del sistema. El par se presenta al modelo junto con la instrucción de determinar si existe contradicción.

Etiqueta	Pares	Porcentaje
Inconsistente (1)	17	57 %
Consistente (0)	13	43 %
Total	30	100 %

Cuadro 3.4: Distribución del dataset V1 (Inconsistencias)

3.2.5. V2 — Evaluación de Testabilidad

El dataset de testabilidad es también una contribución original de este trabajo. Contiene 97 requisitos anotados como testables o no testables, siguiendo los criterios del estándar IEEE 830:

un requisito es testable si puede ser verificado mediante una prueba objetiva, reproducible y cuyos criterios de aceptación son unívocos.

Etiqueta	Muestras	Porcentaje
Testable (1)	60	62 %
No Testable (0)	37	38 %
Total	97	100 %

Cuadro 3.5: Distribución del dataset V2 (Testabilidad)

3.3– Modelos evaluados

3.3.1. Modelos locales (Ollama)

Los modelos locales se ejecutan mediante Ollama, una herramienta que permite gestionar y servir modelos LLM en hardware local mediante una interfaz compatible con la API de OpenAI. Los modelos se cuantizan para reducir el consumo de memoria, utilizando el formato GGUF.

Identificador	Modelo	Parámetros	Cuantización
qwen7b	Qwen 2.5 Coder 7B Instruct	7B	Q5_K_M
llama8b	Llama 3.1 8B Instruct	8B	Q4_K_M
llama3b	Llama 3.2 3B Instruct	3B	Q4_K_M

Cuadro 3.6: Modelos locales evaluados

3.3.2. Modelos de API (NVIDIA NIM)

Los modelos de API se acceden mediante NVIDIA NIM (*NVIDIA Inference Microservices*), una plataforma que proporciona acceso gratuito a modelos LLM de gran tamaño con una interfaz compatible con la API de OpenAI. La elección de NVIDIA NIM frente a OpenAI se justifica por su coste cero, que permite ejecutar el experimento completo sin incurrir en gastos.

Identificador	Modelo	Parámetros
nim.llama70b	meta/llama-3.1-70b-instruct	70B
nim.llama8b	meta/llama-3.1-8b-instruct	8B
nim.mistral	mistralai/mistral-7b-instruct-v0.3	7B

Cuadro 3.7: Modelos de API evaluados (NVIDIA NIM)

3.3.3. Criterios de selección

Los modelos se seleccionaron atendiendo a los siguientes criterios:

- **Representatividad:** Cubrir un amplio rango de tamaños (3B, 7B, 8B, 70B) y familias de modelos (Qwen, Llama, Mistral).
- **Disponibilidad open-source:** Todos los modelos evaluados tienen pesos públicamente disponibles bajo licencias permisivas.

- **Ejecutabilidad local:** Los modelos locales seleccionados pueden ejecutarse en la GPU del equipo de experimentación (NVIDIA RTX 4060, 8 GB VRAM) mediante cuantización.
- **Coste cero:** Los modelos de API se acceden a través de NVIDIA NIM, que proporciona una capa de créditos gratuitos suficientes para el experimento completo.

3.4– Estrategias de prompting

3.4.1. Question Refinement (QR)

La estrategia *Question Refinement* instruye al modelo para que, antes de clasificar el requisito, reformule mentalmente la tarea con el objetivo de aclarar posibles ambigüedades en el enunciado. El prompt sigue el siguiente esquema:

“Clasifica el siguiente requisito [...]. Si la clasificación no es clara, reformula mentalmente la pregunta para entender mejor el requisito antes de clasificar. [...] Clasificación.”

3.4.2. Cognitive Verifier (CV)

El patrón *Cognitive Verifier* pide al modelo que descomponga el problema en pasos de análisis explícitos antes de emitir la respuesta final. Esto se traduce en prompts que enumeran los pasos de razonamiento que el modelo debe seguir.

3.4.3. Persona + Context (PC)

Esta estrategia asigna al modelo una identidad experta en Ingeniería de Requisitos (por ejemplo, “ingeniero de requisitos senior con 10 años de experiencia en IEEE 830 e ISO 29148”) y proporciona contexto sobre las categorías de la tarea, con el objetivo de que el modelo active sus conocimientos sobre el dominio.

3.4.4. Few-Shot (FS)

Se incluyen entre 3 y 5 ejemplos etiquetados en el prompt antes de presentar el caso a clasificar. Los ejemplos se seleccionan de forma fija para garantizar reproducibilidad entre iteraciones. Esta técnica aprovecha el aprendizaje en contexto de los LLMs sin necesidad de ajuste de parámetros.

3.4.5. Chain of Thought (CoT)

La estrategia *Chain of Thought* solicita al modelo que muestre su razonamiento paso a paso antes de emitir la clasificación final. Según Wei y cols. (2022), esta técnica mejora el rendimiento en tareas de razonamiento, especialmente en modelos de mayor tamaño.

3.5– Métricas de evaluación

3.5.1. Métricas de rendimiento

Para todas las tareas se calculan las métricas estándar de clasificación binaria:

Accuracy (Acc): Proporción de predicciones correctas sobre el total.

Precisión (P): Proporción de verdaderos positivos sobre todas las predicciones positivas.

Recall (R): Proporción de verdaderos positivos sobre todos los positivos reales.

F1-score (F1): Media armónica de precisión y recall; métrica principal de comparación en todos los análisis.

La métrica principal de comparación es el **F1-score macro**, que da igual peso a cada clase independientemente de su frecuencia en el dataset.

3.5.2. Tasa de respuestas inválidas (*invalid_rate*)

Se registra la proporción de respuestas del modelo que no pueden ser mapeadas a ninguna etiqueta válida (por ejemplo, respuestas en formato no esperado o en idioma diferente al solicitado). Una tasa alta de respuestas inválidas indica problemas de seguimiento de instrucciones.

3.5.3. Rendimiento computacional

Para los modelos locales se mide:

- **Tokens por segundo (tokens/s):** Velocidad de generación de texto, relevante para evaluar la viabilidad de uso en producción.
- **Latencia media (s):** Tiempo medio de respuesta por consulta.

3.5.4. Análisis estadístico

Para responder a las preguntas de investigación se aplican los siguientes análisis:

- **ANOVA de un factor** para determinar si existen diferencias significativas entre grupos (modelos o estrategias).
- **Prueba t de Student** para comparaciones entre pares específicos.
- **Cohen's d** como estimador del tamaño del efecto.
- Se utiliza un nivel de significación $\alpha = 0,05$ en todas las pruebas.

3.6– Procedimiento experimental

3.6.1. Pipeline de ejecución

El experimento se ejecuta mediante un pipeline automatizado implementado en Python. Para cada configuración (tarea, modelo, estrategia, iteración):

1. Se carga el dataset de la tarea correspondiente.
2. Se barajan las muestras con la semilla de la iteración (seeds: 42, 123, 456, 789, 1024).
3. Para cada muestra se construye el prompt según la estrategia.
4. Se envía el prompt al modelo (local u API) con temperatura 0.4 y máximo 512 tokens.
5. Se parsea la respuesta del modelo para extraer la etiqueta predicha.
6. Se calculan las métricas comparando predicciones con etiquetas reales.
7. Los resultados se almacenan en un fichero CSV junto con metadatos de la ejecución.

3.6.2. Parámetros de generación

Parámetro	Valor
Temperatura	0.4
Máximo de tokens	512
Seeds (iteraciones)	42, 123, 456, 789, 1024
Intentos por fallo	5 (backoff exponencial)

Cuadro 3.8: Parámetros de generación del modelo

3.6.3. Gestión de fallos y reproducibilidad

El pipeline implementa un mecanismo de checkpoint que guarda el progreso al finalizar cada configuración, permitiendo reanudar el experimento ante interrupciones sin perder resultados previos. Los reintentos ante fallos de red o de la API se realizan con un retardo exponencial (*exponential backoff*) de base 2 segundos, con un máximo de 5 intentos por consulta.

3.6.4. Hardware utilizado

Los experimentos locales se ejecutan en el siguiente equipo:

Componente	Especificación
CPU	Intel Core i7-13650HX (20 núcleos, 4.9 GHz)
GPU	NVIDIA GeForce RTX 4060 Max-Q (8 GB VRAM)
RAM	32 GB DDR5
Almacenamiento	SSD NVMe
Sistema operativo	Fedora Linux 43 (kernel 6.18)

Cuadro 3.9: Hardware del entorno de experimentación

Los experimentos con modelos de API (NVIDIA NIM) no dependen del hardware local, ya que la inferencia se realiza en los servidores de NVIDIA.

Sistema desarrollado

Este capítulo describe la arquitectura e implementación del sistema desarrollado para dar soporte al experimento y a la aplicación práctica de los resultados. El sistema se compone de cinco módulos principales: el pipeline de experimentación, el módulo de prompts, los wrappers de modelos, el motor de análisis y un componente de interfaz de usuario.

4.1– Arquitectura general

El sistema sigue una arquitectura en capas que separa claramente las responsabilidades:

Capa de datos: Gestión de los datasets de las cinco tareas en formato CSV, con carga, filtrado y muestreo controlado por semilla.

Capa de modelos: Wrappers que abstraen la comunicación con modelos locales (Ollama) y de API (NVIDIA NIM), con una interfaz uniforme.

Capa de prompts: Generación y parametrización de los 25 prompts (5 tareas \times 5 estrategias), con parsers específicos por tarea para extraer etiquetas de las respuestas en texto libre.

Capa experimental: Orquestación del diseño factorial, gestión de checkpoints y cálculo de métricas.

Capa de análisis: Generación de estadísticas descriptivas, gráficos comparativos y pruebas de significación estadística.

Capa de interfaz: API REST (FastAPI) y frontend interactivo (Streamlit) para uso en modo de análisis individual.

Las tecnologías principales utilizadas son Python 3.14, pandas y NumPy para la gestión de datos, scikit-learn para las métricas, scipy para los análisis estadísticos, matplotlib y seaborn para las visualizaciones, FastAPI para la API REST, y Streamlit para el frontend.

4.2– Pipeline de experimentación (`experiment.py`)

4.2.1. Clase `ExperimentRunner`

La clase `ExperimentRunner` encapsula toda la lógica del experimento. Su constructor recibe la ruta al fichero de configuración `experiment_config.yaml`, que define los modelos,

estrategias, tareas y semillas a utilizar. Los directorios de salida (resultados, checkpoints, logs) se crean automáticamente si no existen.

El método principal `run_experiment(task, models, strategies)` itera sobre el producto cartesiano de modelos, estrategias e iteraciones, ejecutando cada configuración secuencialmente. Para cada muestra del dataset se:

1. Construye el prompt mediante el módulo `prompts.py`.
2. Llama al modelo y mide el tiempo de respuesta.
3. Parsea la respuesta para obtener la etiqueta predicha.
4. Acumula los resultados en un buffer.

Al finalizar cada configuración (modelo, estrategia, iteración), los resultados se guardan en un checkpoint JSON para permitir la reanudación del experimento ante interrupciones.

4.2.2. Mecanismo de checkpoint y reanudación

El sistema mantiene un fichero de checkpoint por experimento con el siguiente formato:

- Listado de configuraciones ya completadas (identificadas por `model+strategy+seed`).
- Resultados parciales acumulados.
- Metadatos del experimento (timestamp de inicio, versión del pipeline).

Al ejecutar el experimento con el flag `--resume`, el pipeline comprueba el checkpoint y omite las configuraciones ya completadas, reanudando a partir de la primera configuración pendiente. Al finalizar el experimento completo, el checkpoint se elimina y los resultados se consolidan en un único CSV.

4.2.3. Modo dry-run

Para verificar el correcto funcionamiento del pipeline sin consumir los recursos completos del experimento, se implementó un modo `--dry-run` que ejecuta solo 5 muestras por configuración y una única iteración con el primer modelo disponible. Este modo se empleó para validar cada nueva tarea o modificación del código antes de lanzar el experimento completo.

4.3– Módulo de prompts (`prompts.py`)

4.3.1. Estructura de los prompts

El módulo `prompts.py` define un diccionario anidado `PROMPTS[task][strategy]` que contiene los 25 templates de prompt como cadenas con la variable `{requirement}` (o `{requirement1}` y `{requirement2}` para la tarea de inconsistencias).

Los templates utilizan Python format strings, lo que permite su parametrización dinámica con el texto del requisito a analizar.

4.3.2. Función `build_prompt`

La función `build_prompt(task, strategy, sample)` recibe el identificador de la tarea, la estrategia y la muestra del dataset, y devuelve el prompt listo para enviar al modelo. Internamente selecciona el template adecuado e inyecta los campos del requisito.

4.3.3. Parsers de respuesta por tarea

Cada tarea tiene un parser específico que extrae la etiqueta predicha del texto libre generado por el modelo:

- **A1 (F/NF):** Busca las letras “F” ó “NF” en la respuesta, priorizando las primeras ocurrencias y filtrando falsos positivos mediante expresiones regulares.
- **A2 (Ambigüedad), A3 (Compleitud), V2 (Testabilidad):** Buscan palabras clave binarias (“YES”/“NO”, “AMBIGUO”/“CLARO”, “1”/“0”) con normalización de mayúsculas.
- **V1 (Inconsistencias):** igual esquema; etiquetas de salida: INCONSISTENTE o CONSISTENTE.

Cuando el parser no encuentra ninguna etiqueta reconocible, la predicción se marca como inválida y se contabiliza en la tasa `invalidrate`.

4.3.4. Prompts combinados y pipeline de documentos

Además de los prompts individuales por tarea, el módulo define un conjunto de **prompts combinados** que solicitan al modelo analizar un requisito en las cuatro dimensiones de calidad (ambigüedad, completitud, inconsistencia, testabilidad) en una única llamada. Este diseño permite al frontend analizar documentos de requisitos de forma eficiente, reduciendo el número de llamadas al modelo.

4.4– Wrappers de modelos (`models.py`)

4.4.1. Interfaz unificada

Todos los modelos exponen el mismo método `generate(prompt, max_tokens)` que devuelve un diccionario con los campos:

- `response`: texto generado por el modelo.
- `tokens_per_second`: velocidad de generación (solo para modelos locales).
- `latency`: tiempo total de la llamada en segundos.
- `model`: nombre del modelo utilizado.

4.4.2. OllamaModel

La clase `OllamaModel` utiliza la biblioteca cliente de Ollama para Python. Los parámetros de generación (temperatura, número máximo de tokens) se configuran en el constructor. El método `generate` mide la latencia con `time.time()` y extrae las métricas de rendimiento del objeto de respuesta de Ollama.

4.4.3. NVIDIANIMModel

La clase `NVIDIANIMModel` utiliza el cliente de OpenAI apuntando al endpoint de NVIDIA NIM (<https://integrate.api.nvidia.com/v1>). La API Key se carga desde el fichero `.env` mediante la biblioteca `python-dotenv`. El modelo no reporta tokens/s, ya que la infraestructura de NVIDIA no expone esta métrica.

4.4.4. Reintentos con backoff exponencial

La función `_retry_with_backoff(func, max_retries=5)` envuelve cualquier llamada al modelo con un mecanismo de reintentos:

- Reintento inmediato ante fallos genéricos con un retardo de 2^{intento} segundos.
- Retardo mínimo de $3 \times \text{intento}$ segundos ante errores HTTP 429 (límite de tasa).
- Si se agotan los 5 intentos, se propaga la última excepción.

Este mecanismo garantiza la robustez del pipeline ante interrupciones de red o límites de tasa de la API.

4.5– Módulo de análisis (`analysis.py`)

El módulo `analysis.py` proporciona funciones para generar informes y visualizaciones a partir de los ficheros CSV de resultados.

4.5.1. Gráficos generados

- **Heatmap modelo × estrategia:** Mapa de calor del F1-score para cada combinación de modelo y estrategia, permitiendo identificar visualmente las mejores configuraciones.
- **Boxplots por modelo:** Distribución del F1-score entre las 5 iteraciones para cada modelo, mostrando variabilidad.
- **Gráfico de barras por tarea:** Comparación del F1-score medio por tarea.

4.5.2. Análisis estadístico

El módulo implementa las siguientes funciones de análisis:

- `run_anova(df, group_col, metric)`: ANOVA de un factor sobre las métricas, agrupadas por modelo o estrategia.
- `run_ttest(df, group1, group2, metric)`: Prueba t de Welch para comparaciones entre pares.
- `cohens_d(group1, group2)`: Estimador del tamaño del efecto.

4.5.3. Generación de informes consolidados

La función `generate_all_reports(results_dir, output_dir)` recorre el directorio de resultados, concatena todos los CSV de la misma tarea (independientemente del origen, local o API) y genera un informe completo con tablas de resumen y gráficos para cada tarea.

4.6– API REST (`api.py`)

4.6.1. Descripción general

La API REST implementada con FastAPI expone los modelos LLM del sistema para su uso desde cualquier cliente HTTP. Su objetivo principal es servir como componente de integración para proyectos externos que deseen incorporar análisis de requisitos sin ejecutar el pipeline experimental completo.

4.6.2. Endpoints principales

Ruta	Método	Descripción
/classify	POST	Clasifica un requisito como F o NF
/analyze	POST	Analiza ambigüedad, completitud o testabilidad
/validate	POST	Detecta inconsistencias entre un par de requisitos
/analyze-all	POST	Análisis combinado de los cuatro atributos de calidad
/health	GET	Comprobación de estado del servicio

Cuadro 4.1: Endpoints de la API REST

4.6.3. Validación y documentación

Todos los modelos de datos de entrada y salida se definen mediante Pydantic, lo que garantiza la validación automática de tipos y la generación de documentación OpenAPI (accesible en /docs) sin configuración adicional.

El middleware CORS está habilitado para permitir el acceso desde el frontend Streamlit y desde cualquier otro cliente web.

4.7– Frontend Streamlit (app.py)

4.7.1. Descripción general

El frontend interactivo implementado con Streamlit permite explorar las capacidades del sistema sin necesidad de conocimientos de programación. Se accede mediante navegador web tras ejecutar `streamlit run app.py`.

4.7.2. Páginas del frontend

El frontend está organizado en siete páginas accesibles desde el menú lateral:

1. **Pipeline Documento:** Permite cargar un documento de texto con requisitos, extraer automáticamente los requisitos mediante un LLM, y analizar cada uno en las cuatro dimensiones de calidad (ambigüedad, completitud, testabilidad, inconsistencias). Los resultados se presentan en una tabla interactiva con codificación de color por semáforo.
2. **Clasificar Requisito (F/NF):** Interfaz para clasificar un requisito individual como Funcional o No Funcional. Permite seleccionar el modelo y la estrategia de prompting, y muestra la respuesta completa del modelo junto con la etiqueta inferida.
3. **Analizar Calidad:** Análisis de un requisito individual en sus atributos de ambigüedad, completitud y testabilidad, con explicación del razonamiento del modelo.
4. **Validar Consistencia:** Verificación de la consistencia entre dos requisitos. El usuario introduce ambos requisitos y el sistema determina si existe contradicción.
5. **Resultados Experimentos:** Dashboard de visualización de los resultados experimentales almacenados en el directorio `results/`. Muestra heatmaps, boxplots y tablas de resumen filtradas por tarea.

6. **Comparar Modelos:** Tabla comparativa del rendimiento de todos los modelos sobre todas las tareas, con posibilidad de exportar a CSV.
7. **Monitorización:** Panel de estado del sistema que muestra los modelos disponibles en Ollama, los recursos del sistema (CPU, RAM, GPU) y el historial reciente de solicitudes.

4.7.3. Sistema de advertencias

El módulo `warnings_analysis.py` implementa un sistema que analiza en tiempo real la calidad de los requisitos introducidos por el usuario antes de su envío al modelo, detectando patrones comunes de baja calidad (términos vagos, ausencia de sujeto, condiciones incompletas). Las advertencias se muestran como mensajes de aviso en la interfaz para guiar al usuario en la mejora del requisito.

Resultados y análisis

Este capítulo presenta los resultados del experimento factorial completo (5 tareas \times 6 modelos \times 5 estrategias \times 5 iteraciones = 750 configuraciones). Para cada tarea se muestran las métricas agregadas, se identifican las combinaciones modelo–estrategia más efectivas y se responde a las tres preguntas de investigación mediante pruebas estadísticas.

5.1– Resultados por tarea

5.1.1. A1 — Clasificación Funcional / No Funcional

La tarea de clasificación es la que mayor rendimiento medio obtiene entre las tareas de análisis, con un F1 global de 0,742. El modelo con mejor desempeño promedio es *qwen7b* (F1 = 0,807), seguido por *nim_mistral* (0,772) y *nim_llama70b* (0,766). La mejor combinación individual es *nim_llama70b* con estrategia *persona_context* (F1 = 0,850, Acc = 0,812).

Modelo	Estrategia	F1	Acc
<i>nim_llama70b</i>	<i>persona_context</i>	0,850	0,812
<i>nim_llama70b</i>	<i>question_refinement</i>	0,838	0,782
<i>llama8b</i>	<i>cognitive_verifier</i>	0,835	0,787
<i>qwen7b</i>	<i>cognitive_verifier</i>	0,834	0,776
<i>qwen7b</i>	<i>persona_context</i>	0,831	0,774

Cuadro 5.1: Mejores resultados en A1 (Clasificación F/NF)

La estrategia *cognitive_verifier* es la que mejor resultado produce de forma consistente (F1 medio entre modelos = 0,785), mientras que *chain_of_thought* es la peor para esta tarea (0,574), probablemente porque induce razonamientos extensos que dificultan la extracción de la etiqueta binaria final.

5.1.2. A2 — Detección de ambigüedad

La detección de ambigüedad es la tarea de análisis más difícil, con un F1 global de 0,627. La mejor combinación individual corresponde a *qwen7b* con *few_shot* (F1 = 0,737), seguida de *nim_llama8b* con *persona_context* (0,732). El 37 % de los requisitos del dataset resultaron difíciles (≥ 50 % de fallos), lo que refleja la naturaleza subjetiva de la ambigüedad.

Modelo	Estrategia	F1	Acc
qwen7b	few_shot	0,737	0,631
nim_llama8b	persona_context	0,732	0,627
nim_mistral	persona_context	0,712	0,572
nim_mistral	cognitive_verifier	0,710	0,576
nim_llama70b	question_refinement	0,681	0,556

Cuadro 5.2: Mejores resultados en A2 (Detección de ambigüedad)

Destaca que `llama8b` con *question_refinement* obtiene $F1 = 0,025$, el peor resultado del experimento en esta tarea, lo que evidencia que la combinación modelo–estrategia influye decisivamente en la capacidad de detección. El modelo `nim_mistral` muestra el mejor rendimiento medio entre los modelos de API para esta tarea ($F1 = 0,704$).

5.1.3. A3 — Evaluación de completitud

La completitud es la tarea más difícil del experimento, con un $F1$ medio global de 0,290. La mitad de los requisitos del dataset (75 / 150) presentan una tasa de fallo superior al 50 %, y dos requisitos fallan con todos los modelos y estrategias (100 % de error), lo que indica problemas de calidad en el propio dataset. La mejor combinación es `qwen7b` con *few_shot* ($F1 = 0,723$), muy por encima de la media, gracias a los ejemplos en el prompt que acotan el criterio de completitud.

Modelo	F1 medio	Tipo
nim_mistral	0,405	API
qwen7b	0,358	Local
llama3b	0,161	Local
llama8b	0,157	Local
nim_llama70b	0,152	API
nim_llama8b	0,106	API

Cuadro 5.3: F1 medio por modelo en A3 (Completitud)

El bajo rendimiento generalizado sugiere que la tarea de completitud requiere razonamiento causal profundo (identificar elementos ausentes) que supera la capacidad actual de los prompts utilizados. La estrategia *few_shot* es la más eficaz ($F1 = 0,395$), seguida de *chain_of_thought* (0,265); los ejemplos en el prompt son especialmente relevantes para definir incompletitud.

5.1.4. V1 — Detección de inconsistencias

La detección de inconsistencias es la segunda tarea más sencilla del experimento, con un $F1$ global de 0,809. Los modelos de API dominan claramente, encabezados por `nim_llama70b` ($F1 = 0,914$). La mejor combinación es `nim_llama70b` con *few_shot* ($F1 = 0,950$, $Acc = 0,940$), seguida muy de cerca por `llama8b` con *chain_of_thought* ($F1 = 0,944$).

Modelo	Estrategia	F1	Acc
nim_llama70b	few_shot	0,950	0,940
nim_llama8b	few_shot	0,935	0,927
nim_llama70b	chain_of_thought	0,935	0,920
llama8b	chain_of_thought	0,944	0,933
qwen7b	chain_of_thought	0,926	0,913

Cuadro 5.4: Mejores resultados en V1 (Detección de inconsistencias)

El dataset de inconsistencias (30 pares) es el de menor tamaño del experimento, lo que favorece resultados más estables y varianzas bajas ($F1\text{ std} < 0,03$). La estrategia *few_shot* es la más efectiva ($F1\text{ medio} = 0,901$), probablemente porque los ejemplos demuestran el tipo de contradicción a identificar.

5.1.5. V2 — Evaluación de testabilidad

La testabilidad es la tarea de validación más difícil y a su vez donde mayor diferencia se observa entre modelos. *nim_llama70b* obtiene el mejor rendimiento global ($F1 = 0,901$), seguido de *qwen7b* (0,878) y *nim_mistral* (0,875). La mejor combinación individual es *nim_llama70b* con *chain_of_thought* ($F1 = 0,916$, $\text{Acc} = 0,895$).

Modelo	Estrategia	F1	Acc
<i>nim_llama70b</i>	<i>chain_of_thought</i>	0,916	0,895
<i>nim_llama70b</i>	<i>cognitive_verifier</i>	0,914	0,899
<i>nim_llama70b</i>	<i>question_refinement</i>	0,911	0,889
<i>qwen7b</i>	<i>chain_of_thought</i>	0,909	0,887
<i>qwen7b</i>	<i>cognitive_verifier</i>	0,908	0,887

Cuadro 5.5: Mejores resultados en V2 (Evaluación de testabilidad)

A diferencia del resto de tareas, el ANOVA de estrategias no revela diferencias significativas para testabilidad ($F = 0,74$, $p = 0,567$), lo que indica que para esta tarea la elección del modelo es el factor determinante, no la estrategia de prompting.

5.2– Comparación local vs. API (RQ1)

El Cuadro 5.6 recoge el F1 medio del grupo de modelos locales (Ollama) y del grupo de modelos de API (NVIDIA NIM) para cada tarea, junto con los resultados de la prueba *t* de Welch y el tamaño del efecto de Cohen.

Tarea	Local	API	Δ	p	d
A1 Clasificación	0,725	0,761	+5,0 %	0,070	−0,30
A2 Ambigüedad	0,550	0,655	+19,1 %	<0,001	−0,68
A3 Completitud	0,225	0,221	−1,8 %	0,900	+0,02
V1 Inconsistencias	0,759	0,861	+13,5 %	<0,001	−0,79
V2 Testabilidad	0,655	0,876	+33,7 %	<0,001	−1,15

Cuadro 5.6: Comparación local vs. API por tarea (F1 medio)

Los resultados muestran que la ventaja de la API no es uniforme. En cuatro de las cinco tareas la API supera a los modelos locales, pero la magnitud varía considerablemente:

- **Clasificación (A1):** La diferencia del 5 % no alcanza significación estadística ($p = 0,070$), con un efecto pequeño ($d = -0,30$). Esto indica que los modelos locales son competitivos para esta tarea concreta.
- **Ambigüedad (A2):** La API supera a los modelos locales en un 19 % ($d = -0,68$, efecto moderado), diferencia significativa.
- **Completitud (A3):** No existe diferencia práctica entre grupos ($d = 0,02$, $p = 0,90$): ninguno de los dos grupos resuelve bien esta tarea.
- **Inconsistencias (V1):** La API supera un 13,5 % con efecto grande ($d = -0,79$).

- **Testabilidad (V2):** La brecha más grande, +34 % ($d = -1,15$); modelos de gran escala son indispensables para esta tarea.

Rango	Modelo	F1 medio	Tipo
1	nim_mistral	0,716	API
2	qwen7b	0,704	Local
3	nim_llama70b	0,681	API
4	nim_llama8b	0,628	API
5	llama8b	0,555	Local
6	llama3b	0,489	Local

Cuadro 5.7: Ranking global de modelos (F1 medio sobre las 5 tareas)

Destacan dos hallazgos relevantes del ranking global: el modelo de API *nim_mistral* (7B parámetros) lidera el ranking con $F1 = 0,716$, y el modelo local *qwen7b* ocupa el segundo puesto (0,704) por delante de *nim_llama70b* (0,681), un modelo diez veces más grande. Esto sugiere que el tamaño del modelo no es el único determinante del rendimiento: la arquitectura y el ajuste fino específico de *qwen7b* para tareas de código e instrucciones le confieren ventajas en varias tareas.

5.3– Comparación por estrategia de prompting (RQ2)

El Cuadro 5.8 muestra el F1 medio de cada estrategia para cada tarea, calculado como la media sobre los seis modelos.

Estrategia	A1	A2	A3	V1	V2
Question Refinement	0,751	0,641	0,206	0,820	0,816
Cognitive Verifier	0,785	0,688	0,210	0,769	0,875
Persona + Context	0,774	0,659	0,254	0,810	0,876
Few-Shot	0,715	0,701	0,395	0,901	0,823
Chain of Thought	0,574	0,628	0,265	0,883	0,879

Cuadro 5.8: F1 medio por estrategia de prompting y tarea

El análisis de la tabla revela que no existe una estrategia universalmente superior. La elección óptima depende de la tarea:

- **A1 (Clasificación):** *Cognitive Verifier* es la mejor estrategia (0,785), significativamente por encima de *Chain of Thought* (ANOVA: $F = 29,14$, $p < 0,001$). La descomposición en preguntas auxiliares ayuda a identificar el tipo funcional.
- **A2 (Ambigüedad):** *Few-Shot* lidera (0,701) gracias a los ejemplos que muestran patrones de ambigüedad concretos.
- **A3 (Complejidad):** *Few-Shot* es claramente la mejor (0,395) frente a las demás ($\leq 0,265$), con diferencias significativas (ANOVA: $F = 5,61$, $p = 0,0003$).
- **V1 (Inconsistencias):** *Few-Shot* lidera (0,901), seguida de *Chain of Thought* (0,883).
- **V2 (Testabilidad):** No hay diferencias significativas entre estrategias (ANOVA: $F = 0,74$, $p = 0,567$); el F1 oscila entre 0,816 y 0,879.

En conjunto, *Few-Shot* es la estrategia más robusta entre tareas: aparece en el top-2 de cuatro de las cinco tareas, siendo especialmente eficaz cuando la tarea requiere identificar un concepto que los LLMs no dominan sin ejemplos (complejidad, inconsistencia).

5.4– Análisis de trade-offs (RQ3)

Además del rendimiento, el diseño experimental captura la latencia media por inferencia y los tokens por segundo generados, lo que permite evaluar la eficiencia (F1 por segundo) de cada modelo.

Modelo	Tipo	F1 medio	Tokens/s	F1/s
nim_mistral	API	0,716	44	0,558
nim_llama8b	API	0,628	122	0,653
nim_llama70b	API	0,681	69	0,458
qwen7b	Local	0,704	38	0,339
llama8b	Local	0,555	44	0,251
llama3b	Local	0,489	82	0,193

Cuadro 5.9: Eficiencia media de los modelos (F1/s y latencia)

Del análisis de trade-offs se extraen tres perfiles diferenciados:

- **Máxima eficiencia:** *nim_llama8b* es el modelo más eficiente (F1/s=0,653) gracias a su alta velocidad de generación (122 tokens/s en promedio) y un F1 aceptable. Adecuado cuando la latencia es un requisito crítico.
- **Mejor balance rendimiento–privacidad:** *qwen7b* combina el segundo mejor F1 global (0,704) con ejecución completamente local, sin enviar datos a servidores externos. Su eficiencia (F1/s=0,339) es inferior a los modelos de API, pero el coste operativo es cero una vez desplegado.
- **Máxima calidad:** *nim_mistral* y *nim_llama70b* ofrecen el mejor rendimiento en tareas complejas, a costa de depender de una API externa, lo que implica costes por uso y exposición de los requisitos (potencialmente confidenciales) a terceros.

Estos tres perfiles constituyen la respuesta a RQ3: la elección óptima depende del contexto organizativo. En entornos con requisitos sensibles o sin acceso a internet, *qwen7b* es la opción recomendada. En entornos donde la latencia o la precisión son críticas, *nim_llama8b* o *nim_llama70b* son preferibles.

5.5– Análisis estadístico

El Cuadro 5.10 resume los resultados del análisis ANOVA de un factor para los factores *modelo* y *estrategia* en cada tarea.

Tarea	<i>F</i> modelo	<i>p</i> modelo	<i>F</i> estrat.	<i>p</i> estrat.
A1 Clasificación	11,04	<0,001	29,14	<0,001
A2 Ambigüedad	11,78	<0,001	12,56	<0,001
A3 Completitud	12,38	<0,001	5,61	<0,001
V1 Inconsistencias	14,10	<0,001	5,84	<0,001
V2 Testabilidad	66,96	<0,001	0,74	0,567

Cuadro 5.10: Resultados ANOVA de un factor por tarea (métrica: F1)

En todas las tareas el factor *modelo* tiene un efecto significativo ($p < 0,001$), con el estadístico *F* más elevado en testabilidad ($F = 66,96$), donde la variabilidad entre modelos es extrema: *llama3b* obtiene $F1 = 0,378$ frente a 0,901 de *nim_llama70b*. El factor *estrategia* es significativo en cuatro de las cinco tareas; la excepción es V2 (testabilidad), donde la selección del modelo es el único determinante relevante del rendimiento.

La comparación local-API mediante la prueba t de Welch revela diferencias significativas en tres tareas (A2, V1, V2), con tamaños del efecto moderado-grande según la clasificación de Cohen ($|d| > 0,5$). En A1 el efecto es pequeño y no significativo, y en A3 ambos grupos son estadísticamente equivalentes.

Conclusiones

Este capítulo sintetiza los hallazgos del experimento factorial, responde a las tres preguntas de investigación planteadas en el Capítulo 1, enumera las contribuciones del trabajo y señala sus limitaciones y posibles líneas futuras.

6.1– Respuesta a las preguntas de investigación

6.1.1. RQ1 — ¿En qué medida difieren los modelos locales y los modelos de API en tareas de Ingeniería de Requisitos?

La respuesta no es uniforme: la ventaja de los modelos de API depende de la tarea.

En **clasificación funcional/no funcional** (A1), la diferencia es pequeña (+5 %, $d = -0,30$) y estadísticamente no significativa ($p = 0,070$), lo que indica que los modelos locales son viables para esta tarea. En **evaluación de completitud** (A3), ambos grupos obtienen rendimientos equivalentes y bajos ($F1 \approx 0,22$), lo que refleja que la tarea es difícil para todos los modelos independientemente del tipo de acceso.

En cambio, para **detección de ambigüedad** (A2, $\Delta = +19$ %), **inconsistencias** (V1, +14 %) y **testabilidad** (V2, +34 %), la API supera significativamente a los modelos locales ($p < 0,001$), con efectos moderados a muy grandes (d entre $-0,68$ y $-1,15$).

El modelo local más competitivo, `qwen7b` ($F1$ global = 0,704), supera en el ranking general a `nim_llama70b` (0,681), un modelo diez veces mayor, lo que demuestra que la arquitectura y el ajuste fino pueden compensar parcialmente la diferencia de tamaño. Sin embargo, en las tareas de validación más complejas (V1 y V2), la brecha resulta insalvable sin acceder a modelos de gran escala.

6.1.2. RQ2 — ¿Qué estrategia de prompting produce el mejor rendimiento?

No existe una estrategia universalmente superior: la elección óptima depende de la naturaleza de la tarea.

Few-Shot es la estrategia más robusta entre tareas, siendo la mejor o segunda mejor en cuatro de las cinco tareas. Resulta especialmente eficaz en tareas donde el concepto a evaluar es difícil de comunicar sin ejemplos: completitud ($F1$ medio = 0,395, frente a $\leq 0,265$ de las demás) e inconsistencias (0,901).

Cognitive Verifier es la estrategia óptima para clasificación ($F1 = 0,785$), probablemente porque la descomposición en preguntas auxiliares obliga al modelo a razonar explícitamente sobre los criterios funcional/no funcional.

Para **testabilidad**, las diferencias entre estrategias no son significativas (ANOVA: $p = 0,567$): la selección del modelo importa mucho más que la estrategia empleada.

Chain of Thought es la peor estrategia para clasificación ($F1 = 0,574$), pero compite con *Few-Shot* en tareas de validación, donde el razonamiento explícito ayuda a detectar contradicciones o criterios de verificabilidad. CoT resulta perjudicial cuando el razonamiento extenso dificulta la extracción de la etiqueta final.

6.1.3. RQ3 — ¿Cuáles son los trade-offs entre rendimiento, coste y privacidad?

El experimento identifica tres perfiles claramente diferenciados:

- **Máximo rendimiento:** `nim_llama70b` (API) ofrece el mejor resultado en tareas de validación (V2: $F1 = 0,916$), pero requiere enviar los requisitos a una infraestructura externa y depende de la disponibilidad de la API.
- **Mejor balance rendimiento–privacidad:** `qwen7b` (local) alcanza un $F1$ global de 0,704, el segundo mejor del experimento, con ejecución completamente privada y coste nulo tras el despliegue. Recomendado para organizaciones con requisitos confidenciales o entornos sin acceso a internet.
- **Mayor eficiencia:** `nim_llama8b` (API) ofrece el mejor ratio $F1$ /latencia (0,653 $F1/s$) gracias a su alta velocidad (122 tokens/s), con un $F1$ global de 0,628. Recomendado cuando la latencia en tiempo real es un requisito crítico.

6.2– Contribuciones del trabajo

Este trabajo aporta las siguientes contribuciones originales al campo de la Ingeniería de Requisitos asistida por LLMs:

- **Primera evaluación sistemática de modelos LLM locales en IR.** Hasta la fecha, ningún estudio de la literatura revisada por Cheng y cols. (2024) evalúa modelos ejecutados localmente. Este trabajo cubre tres modelos locales (Qwen 2.5 7B, Llama 3.1 8B, Llama 3.2 3B) en cinco tareas de IR, estableciendo una línea base de referencia para la comunidad.
- **Diseño factorial completo de 750 configuraciones.** El experimento es, hasta donde se conoce, el más amplio en términos de combinaciones modelo–estrategia evaluadas simultáneamente en tareas de IR, con análisis estadístico riguroso (ANOVA, prueba t de Welch, Cohen’s d).
- **Datasets anotados para completitud y testabilidad.** Los datasets de las tareas A3 y V2 son contribuciones originales de este trabajo, dado que no existen conjuntos de datos públicos para estas tareas en el contexto de IR. Ambos datasets están disponibles en el repositorio del proyecto.
- **Sistema integrado de experimentación y análisis.** El proyecto incluye un pipeline experimental con soporte de checkpoint y reanudación, un módulo de análisis estadístico automatizado, una API REST y un frontend interactivo, formando un sistema completo y reutilizable para futuras evaluaciones.
- **Limpieza y normalización de datasets públicos.** Se ha documentado y corregido la presencia de caracteres de control, saltos de línea y caracteres Unicode no normalizados en los datasets utilizados, incluyendo el dataset PROMISE NFR (Cleland-Huang y cols. (2007)), mejorando su calidad para uso futuro.

6.3– Limitaciones

- **Tamaño de los datasets propios.** Los datasets de completitud (150 muestras) y testabilidad (97 muestras) son de tamaño reducido, lo que limita la generalización estadística de los resultados en estas tareas. El análisis de errores muestra que en completitud el 50 % de los requisitos presenta tasas de fallo superiores al 50 %, lo que sugiere tanto dificultad intrínseca de la tarea como posibles problemas de anotación en el dataset.
- **Calidad del dataset PROMISE NFR.** Se han detectado 52 registros problemáticos en `promise_nfr_v2.csv`: 8 textos de menos de 20 caracteres (etiquetas en lugar de requisitos completos), 36 con saltos de línea internos y 8 con caracteres de control Windows heredados del dataset original. Estos errores, presentes en el dataset fuente Cleland-Huang y cols. (2007), pueden haber contribuido a los fallos en cascada observados en algunos requisitos.
- **Hardware local limitado.** La GPU disponible (NVIDIA RTX 4060 Max-Q, 8 GB VRAM) restringe los modelos locales evaluados a un máximo de 8B parámetros con cuantización Q4 o Q5. Modelos locales de mayor tamaño (13B, 34B) podrían reducir la brecha con la API.
- **Obsolescencia acelerada de modelos.** Los modelos evaluados corresponden a versiones disponibles hasta principios de 2025. La rápida evolución del campo puede hacer que las conclusiones cuantitativas queden desactualizadas en el corto plazo, aunque las conclusiones metodológicas mantienen su validez.
- **Ausencia de validación humana.** Las métricas se calculan respecto a etiquetas anotadas por los autores, sin validación inter-anotador para los datasets propios (A3 y V2). Un coeficiente de acuerdo entre anotadores (kappa de Cohen) fortalecería la credibilidad de las anotaciones.

6.4– Trabajo futuro

- **Ampliar y validar los datasets.** Sustituir los datasets sintéticos por corpus públicos con validación inter-anotador, en particular integrando ReqEval para la tarea de ambigüedad y el dataset completo PROMISE_exp (969 muestras) para clasificación.
- **Evaluar modelos de mayor tamaño localmente.** Aprovechar técnicas de cuantización más agresiva (GGUF Q2, Q3) o hardware adicional para evaluar modelos de 13B, 34B y 70B ejecutados localmente, reduciendo la brecha con la API sin comprometer la privacidad.
- **Ajuste fino (*fine-tuning*) sobre datasets de IR.** Comparar el rendimiento del prompting evaluado en este trabajo con modelos ajustados específicamente para tareas de IR, lo que permitiría cuantificar el valor añadido del fine-tuning frente a la ingeniería de prompts.
- **Extender el análisis a nuevas tareas de IR.** Incorporar tareas no evaluadas en este trabajo: trazabilidad de requisitos, priorización, detección de duplicados y verificación de conformidad con estándares (IEEE 830, ISO 29148).
- **Integrar el sistema con herramientas industriales de gestión de requisitos como JIRA o IBM DOORS.**
- **Análisis de incertidumbre y calibración.** Evaluar si los modelos están calibrados, es decir, si su nivel de confianza en la predicción correlaciona con su tasa de acierto, lo que permitiría filtrar predicciones de baja confianza en entornos de producción.

Referencias

- Boehm, B. W., y Papaccio, P. N. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10), 1462–1477. doi: 10.1109/32.6191
- Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... others (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 33, 1877–1901.
- Cheng, H., Husen, J. H., Lu, Y., Racharak, T., Yoshioka, N., Ubayashi, N., y Washizaki, H. (2025). Generative AI for requirements engineering: A systematic literature review. *Software: Practice and Experience*. (Preprint: arXiv:2409.06741. VERIFICAR: confirmar que este es el paper de referencia y actualizar el dato “0 % de estudios evalúa modelos locales”) doi: 10.1002/spe.70029
- Cleland-Huang, J., Settini, R., Zou, X., y Solc, P. (2007). Automated classification of non-functional requirements. En *Requirements engineering* (Vol. 12, pp. 103–120). Springer. (Dataset PROMISE NFR) doi: 10.1007/s00766-007-0045-1
- Femmer, H., Méndez Fernández, D., Wagner, S., y Eder, S. (2017). Rapid quality assurance with requirements smells. *Journal of Systems and Software*, 123, 190–213. doi: 10.1016/j.jss.2016.02.047
- Ronanki, K., Cabrero-Daniel, B., Horkoff, J., y Berger, C. (2024). Requirements engineering using generative AI: Prompts and prompting patterns. En A. Nguyen-Duc, P. Abrahamsson, y F. Khomh (Eds.), *Generative AI for effective software development* (pp. 109–127). Springer Nature Switzerland. (Preprint: arXiv:2311.03832) doi: 10.1007/978-3-031-55642-5_5
- Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. En *Advances in neural information processing systems (neurips)* (Vol. 30, pp. 5998–6008).
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., ... Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems (NeurIPS)*, 35, 24824–24837.
- White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., ... Schmidt, D. C. (2023). *A prompt pattern catalog to enhance prompt engineering with ChatGPT*. Descargado de <https://arxiv.org/abs/2302.11382>