Code Review Automation Leveraging LLMs

Joaquín Olmos[1], Juan Cruz Gardey[1], and Julián Grigera[1,2,3]

[1] LIFIA, Fac. de Informatica, Univ. Nac. de La Plata, La Plata, Argentina
{jolmos,juliang,jcgardey}@lifia.info.unlp.edu.ar
[2] CONICET, Argentina
[3] CICPBA, Pcia. de Buenos Aires, Argentina

Abstract. In modern software development, agile methods demand continuous delivery, increasing pressure on development teams and exposing inefficiencies in time-consuming tasks such as manual code review. Although essential for ensuring software quality, code review is often perceived as costly and burdensome, leading many teams to either minimize its use or struggle to sustain it effectively within their workflows. Recent advances in Large Language Models (LLMs) have enabled automated approaches to code review; however, current research primarily focuses on advancing and specializing models through pre-training or fine-tuning. While effective, these approaches require substantial computational resources, time investment, and domain expertise, limiting their applicability in real-world scenarios. This work proposes a lightweight and practical alternative for partially automating code review using off-the-shelf LLMs, without additional training. The solution leverages prompt engineering and API-based access to integrate seamlessly into CI/CD pipelines, generating structured feedback on pull requests based on configurable rules. A threefold evaluation demonstrates consistent performance across varying input sizes and shows that the tool reduces developer workload by acting as a preliminary filter. This approach provides an accessible, scalable solution aligned with human-centered AI principles.

Keywords: code review automation, large language models (LLMs), prompt engineering, continuous integration and continuous delivery (CI/CD), human-centered AI, static code analysis

Automatización de Code Review utilizando LLMs

Resumen En el desarrollo moderno de software, los métodos ágiles exigen una entrega rápida y continua, lo que incrementa la presión sobre los equipos de desarrollo y pone en evidencia ineficiencias en tareas que consumen mucho tiempo, como la revisión manual de código. Aunque es fundamental para garantizar la calidad del software, la revisión de código suele percibirse como costosa y demandante, lo que lleva a muchos equipos a minimizar su uso o a tener dificultades para sostenerla

de manera efectiva dentro de sus flujos de trabajo. Los avances recientes en los Modelos de Lenguaje de Gran Escala (LLMs) han habilitado enfoques automatizados para la revisión de código; sin embargo, la investigación actual se centra principalmente en el avance y la especialización de los modelos mediante preentrenamiento o ajuste fino. Si bien estos enfoques son efectivos, requieren una cantidad considerable de recursos computacionales, inversión de tiempo y conocimiento especializado, lo que limita su aplicabilidad en escenarios reales. Este trabajo propone una alternativa liviana y práctica para la automatización parcial de la revisión de código utilizando modelos LLM disponibles "off-the-shelf", es decir, sin necesidad de entrenamiento adicional. La solución aprovecha técnicas de ingeniería de prompts y el acceso a través de APIs para integrarse de manera fluida en pipelines de CI/CD, generando retroalimentación estructurada sobre pull requests basada en reglas configurables. Una evaluación en tres dimensiones demuestra un rendimiento consistente frente a distintos tamaños de entrada y evidencia el potencial de la herramienta para reducir carga de trabajo al actuar como filtro preliminar. Este enfoque ofrece una solución accesible y escalable, alineada con principios de inteligencia artificial centrada en el ser humano.

Keywords: automatización de revisión de código, modelos de lenguaje de gran escala (LLMs), ingeniería de prompts, integración continua y entrega continua (CI/CD), inteligencia artificial centrada en el humano, análisis estático de código

1. Introducción

En la industria del desarrollo de software, los métodos ágiles se han consolidado como un estándar ampliamente adoptado. Según Licorish et al. (2016), el 71,2 % de las organizaciones entrevistadas reportan utilizar Scrum en alguna etapa del ciclo de vida del software. Estas metodologías priorizan la entrega continua de valor al cliente mediante prácticas como reuniones diarias (*stand-ups*), retrospectivas y planificación iterativa. No obstante, este énfasis en la entrega continua ha puesto en evidencia una problemática recurrente: el tiempo invertido en tareas auxiliares, particularmente la revisión manual de código (*code review*). En este contexto, Bosu & Carver (2013) señalan que los desarrolladores senior dedican en promedio 6,4 horas diarias a revisar código en proyectos de gran escala, lo que reduce significativamente su disponibilidad para actividades de mayor valor agregado. A su vez, las herramientas tradicionales de revisión automatizada, como SonarQube o Checkstyle, presentan limitaciones en términos de flexibilidad y adaptabilidad, dificultando su integración efectiva en equipos Scrum con dinámicas y necesidades específicas.

Frente a este escenario, la irrupción de la inteligencia artificial generativa (*genAI*) abrió nuevas oportunidades para repensar los procesos de revisión de

código. Modelos como ChatGPT o Claude han demostrado capacidades avanzadas de razonamiento y adaptación en lenguaje natural, posicionándose como candidatos prometedores para asistir en tareas de análisis y revisión de código. Estudios recientes evidencian mejoras en la exactitud de detección de cambios mediante técnicas de preentrenamiento de modelos de lenguaje de gran escala (*LLMs*) (Li et al., 2022).

Como consecuencia de los avances en inteligencia artificial, la automatización de la revisión de código ha evolucionado significativamente, desde herramientas de análisis estático tradicionales hacia enfoques basados en aprendizaje profundo, modelos preentrenados a gran escala y arquitecturas multiagente orientadas tanto a la generación de comentarios como a la resolución automática de observaciones (Li et al., 2022) (Frömmgen et al., 2024). Sin embargo, estudios comparativos indican que para alcanzar un rendimiento óptimo, estas soluciones dominantes suelen requerir arquitecturas complejas o algún grado de sintonización fina (fine-tuning) específica (Lu et al. 2023) (Pornprasit y Tantithamthavorn, 2024). Esto introduce altos costos de entrenamiento y un persistente problema de "arranque en frío" ante la falta de datos etiquetados, volviendo a estos enfoques poco accesibles para equipos con recursos limitados o plazos acotados.

En este contexto, el presente trabajo se posiciona proponiendo un enfoque liviano basado exclusivamente en prompt engineering y en el consumo de modelos off-the-shelf a través de APIs. Al aprovechar la capacidad de razonamiento nativa de los LLMs modernos mediante estrategias estructuradas de prompting, nuestra propuesta elimina la necesidad de reentrenamientos masivos y facilita una adopción económica e inmediata dentro de flujos de trabajo ágiles y líneas de CI/CD. Siguiendo los principios de Human-Centered AI de Ben Shneiderman (2022), el objetivo de este enfoque no es reemplazar el criterio del revisor humano, sino brindarle una herramienta ágil que funcione como un primer filtro preliminar previo a su intervención. Esta perspectiva se apoya en el principio de "Humans in the group; computers in the loop", el cual sostiene que las personas deben mantener el control sobre las herramientas tecnológicas y asumir la responsabilidad por sus resultados. En consecuencia, la automatización propuesta no delega autonomía total a los sistemas de IA, sino utilizarlos como un soporte estratégico para mitigar la carga cognitiva inicial, permitiendo que los desarrolladores concentren sus esfuerzos en tareas de mayor valor.

La herramienta desarrollada en este trabajo se integra de manera orgánica al flujo de integración continua y entrega continua (*CI/CD*). En particular, la revisión automática se ejecuta cuando un cambio de código es propuesto para su incorporación a la rama principal. La herramienta no decide sobre la aceptación o el rechazo del código, sino que genera sugerencias que simplifican la tarea del revisor humano, quien conserva la responsabilidad final sobre la aprobación o el rechazo de la integración.

Finalmente, los resultados de esta investigación muestran que es posible reducir la sobrecarga de los desarrolladores en tareas auxiliares como el *code review*, al mismo tiempo que se facilita la incorporación de revisión automática en entornos ágiles. Asimismo, la herramienta está preparada para maximizar la consistencia

en las respuestas para un mismo *input* mediante la configuración del modelo (por ejemplo, bajando su temperatura). De esta manera, se presenta una alternativa práctica que aprovecha las capacidades de los *LLMs* sin desviar el foco central del problema abordado: la mejora de la eficiencia en los equipos de desarrollo.

2. Trabajo Relacionado

El desarrollo reciente de LLMs ha permitido automatizar la revisión automática de código. Esta tarea es particularmente relevante dado su impacto en la calidad del software, la productividad de los equipos de desarrollo y la detección temprana de errores. En algunos casos se han desarrollado nuevos modelos entrenados para revisión de código, mientras que otros se enfocan en adaptar modelos preentrenados. En esta sección presentamos primero conceptos asociados a esta tarea, para luego situar la presente investigación dentro del estado del arte.

2.1. Prompting

Actualmente, la revisión automática de código asistida por IA se basa en entrenar modelos para este fin (Rasheed et al., 2025), o bien adaptar modelos preentrenados (Lu et al. 2023). Nuestra propuesta, en cambio, se basa en utilizar modelos existentes para evitar entrenamientos adicionales costosos. Con esto buscamos que equipos de trabajo con conocimientos limitados en *Machine Learning* y, en particular, en LLMs, puedan configurar y utilizar la herramienta sin grandes barreras técnicas.

Los trabajos más cercanos a nuestro enfoque son los que desarrollan técnicas avanzadas de prompting. El *zero-shot learning*, por ejemplo, consiste en generar una salida a partir de una instrucción y una entrada sin necesidad de ejemplos adicionales (Pornprasit & Tantithamthavorn, 2024). Aunque no es la forma más sofisticada de prompting, permite abordar casos en los que ejemplificar un resultado puede resultar nocivo para las respuestas producidas por el LLM, como cuando el rango de respuestas es demasiado amplio para ser comprendido en textos acotados. El *few-shot learning*, en cambio, introduce un conjunto reducido de ejemplos de demostración junto con las instrucciones. Esta técnica ha demostrado mejorar significativamente el rendimiento frente al *zero-shot learning*, al ofrecer un contexto que orienta mejor la generación de respuestas. Yendo a estrategias más avanzadas, el *Chain-of-Thought* (CoT) prompting, si bien se ha consolidado para tareas de razonamiento complejo (como las aritméticas o simbólicas) sus beneficios en la generación de código resultan más limitados en comparación con el *few-shot prompting* (Wei et al. 2023). El *Structured Chain-of-Thought* (SCoT) prompting, sin embargo, ha introducido una mejora sustancial al estructurar los pasos intermedios del razonamiento mediante secuencias, bifurcaciones y bucles (Li et al. 2023), incrementando la precisión en la generación de código y elevando la calidad percibida por desarrolladores humanos. En este trabajo, el SCoT se aprovecha para ordenar el proceso de revisión que el LLM debe realizar en pasos ordenados y repetibles.

Otra línea relevante es la del *Pattern-Exploiting Training* (PET), una técnica de clasificación de texto en escenarios *few-shot* que combina instrucciones textuales con *fine-tuning* basado en ejemplos. Este enfoque ha alcanzado resultados comparables al desempeño humano en benchmarks de tareas reales, como RAFT, superando incluso a modelos de gran escala como GPT-3. Finalmente, los avances en *Prompt Composition* y *Self-Refinement* dentro de sistemas multiagente, como el marco FlowGenScrum, integran estrategias de razonamiento encadenado, composición de prompts y auto-refinamiento. Estas combinaciones han logrado mejoras sustanciales tanto en precisión (con un incremento del 15 % en Pass@1 respecto a RawGPT) como en la calidad del código, reduciendo errores estructurales y mejorando el manejo de excepciones.

En síntesis, mientras que los estudios mencionados se concentran en el desarrollo de técnicas de prompting avanzadas o en el entrenamiento adicional de los modelos, la presente investigación se diferencia al proponer una alternativa más accesible y práctica. Su foco está puesto en el aprovechamiento de los modelos a través de API y en la elaboración de guías aplicables para equipos con menor especialización técnica, facilitando así la adopción de estas herramientas en contextos reales de trabajo.

2.2. Revisión de Código

La revisión de código se ha convertido en una práctica esencial orientada a mejorar la calidad del software, garantizando la entrega de productos con menos defectos tanto a nivel estructural como funcional. Esta práctica ha evolucionado y se ha adaptado a los distintos enfoques metodológicos de desarrollo, extendiéndose hoy a la mayoría de los equipos de ingeniería de software. Según un informe de 2022 (TrustRadius, 2022), alrededor del 84 % de los equipos de desarrollo implementan algún tipo de proceso de revisión de código.

En la actualidad, el *code review* suele realizarse luego de aprobar los tests de unidad y antes de integrar a la rama principal de los repositorios. En este punto tiene lugar la revisión por pares (*peer review*). Si el revisor detecta deficiencias, puede emitir observaciones, solicitar correcciones y rechazar la solicitud de unión (*merge request* o *pull request*). Caso contrario, se aprueba la integración del código al repositorio principal (Bosu et al., 2015).

Si bien la revisión de código presenta ventajas evidentes en términos de calidad y mantenimiento, también implica desafíos como el costo en tiempo, debido a la complejidad de analizar grandes volúmenes de código. Según un estudio (Bosu & Carver, 2013), los desarrolladores reportan una dedicación promedio de 6,4 horas diarias a tareas de revisión, por lo cual algunos equipos optan por el *self-review*, es decir, que el propio autor revisa su código antes de integrarlo, perdiendo así los beneficios del control cruzado y la retroalimentación de miembros con mayor nivel de experiencia.

En este contexto nuestra propuesta busca automatizar parcialmente el proceso de revisión de código para reducir la carga de trabajo sobre los desarrolladores, minimizando la dependencia del *self-review* y promoviendo un flujo de

trabajo más eficiente, colaborativo y alineado con los estándares de calidad de la ingeniería de software contemporánea.

2.3. Soluciones Relacionadas

Dentro de las propuestas presentadas por la comunidad investigadora se observa una tendencia a enfatizar el reentrenamiento, ya sea parcial o completo, de los LLMs destinados a la tarea de revisión de código.

Rasheed et al. (2025) entrenan su modelo en grandes repositorios de código, que incluyen revisiones, informes de errores y documentación de mejores prácticas. Por su parte, Li et al.(2022) abordan el preentrenamiento de LLMs a gran escala para la automatización de la revisión de código. Este trabajo menciona como objetivos el *denoising* y el uso de datos bimodales (cambios de código y comentarios de revisión) en la fase de preformación.

Otra posibilidad es el *Parameter Efficient Fine-Tuning* (PEFT), descrita por Lu et al. (2023). Los autores presentan un *framework* que utiliza LLaMA para automatizar la revisión de código. A pesar de emplear la versión base más pequeña de LLaMA, *LLaMA-Reviewer* logra igualar el rendimiento de modelos previamente desarrollados para revisión de código sin requerir un preentrenamiento desde cero, lo cual demanda grandes recursos. El *framework* emplea un proceso de sintonización fina en dos etapas: (1) sintonización mediante instrucciones y (2) sintonización fina supervisada. Se mencionan específicamente técnicas de PEFT como *zero-init attention prefix-tuning* y *Low-Rank Adaptation* (LoRA), en las cuales solo se entrena una parte de los parámetros.

Finalmente, otra línea de solución apunta a atacar el problema desde la raíz, a través del desarrollo de herramientas de asistencia para responder comentarios de revisión de código. Un ejemplo de ello es la propuesta de Frömmgenl et al. (2024), que implicó la curación de un conjunto de datos a partir de millones de revisiones de código en Google, con el fin de entrenar un modelo de *machine learning* especializado. Dicho modelo, basado en *fine-tuning*, se beneficia de la generación de datos sintéticos que amplían las oportunidades de edición.

También se puede mencionar a SonarQube, un servicio dedicado a relevar calidad de código. Inicialmente potenciado con técnicas de expresiones regulares y análisis estático, recientemente se han integrado herramientas de *Machine Learning* que permiten generar sugerencias específicas y con contexto.

Finalmente, el trabajo de Pornprasit et al. (2024) es el más similar a nuestra propuesta. Dicho artículo se enfoca en la sintonización fina (*fine-tuning*) y la ingeniería de prompts para la automatización de la revisión de código basada en LLMs. Explica que el *fine-tuning* implica entrenar el modelo en un conjunto de datos específico de revisión de código, mientras que el *pre-training* de LLMs para esta tarea puede resultar costoso. Se afirma que el *fine-tuning* de GPT-3.5 mejora significativamente el rendimiento, incluso con una cantidad limitada de datos de entrenamiento.

Considerando estos trabajos, nuestra propuesta indaga sobre la posibilidad de hacer uso únicamente de la ingeniería de prompts para producir una solución accesible para equipos de desarrollo sin necesidad de realizar procesos costosos

de reentrenamiento o sintonización fina de los modelos. Este enfoque permite reducir significativamente las barreras técnicas y económicas asociadas al uso de modelos de lenguaje en tareas de revisión de código, facilitando su adopción en entornos de desarrollo reales. Asimismo, la propuesta se orienta a evaluar hasta qué punto la ingeniería de prompts, aplicada de manera sistemática, puede reproducir comportamientos de análisis consistentes y útiles para los desarrolladores, sin recurrir a procesos adicionales de entrenamiento del modelo.

3. Propuesta

La propuesta consiste en automatizar parcialmente el proceso de revisión de código mediante la integración de un agente basado en LLMs dentro del flujo CI/CD. El sistema analiza los cambios introducidos en un *pull request* y los contrasta contra un conjunto de reglas de calidad de código configurables definidas mediante un archivo en formato YAML, permitiendo adaptar el comportamiento del análisis a las necesidades específicas de cada proyecto.

Este enfoque se apoya en la capacidad de los LLMs para procesar grandes volúmenes de código y generar observaciones contextualizadas en tiempos reducidos, posicionándolos como un mecanismo eficiente para asistir en etapas tempranas de validación dentro del ciclo de desarrollo. A diferencia de aproximaciones basadas en entrenamiento específico, la solución delega el control del comportamiento en el diseño de prompts, lo que reduce los costos de adopción y facilita su incorporación en entornos existentes.

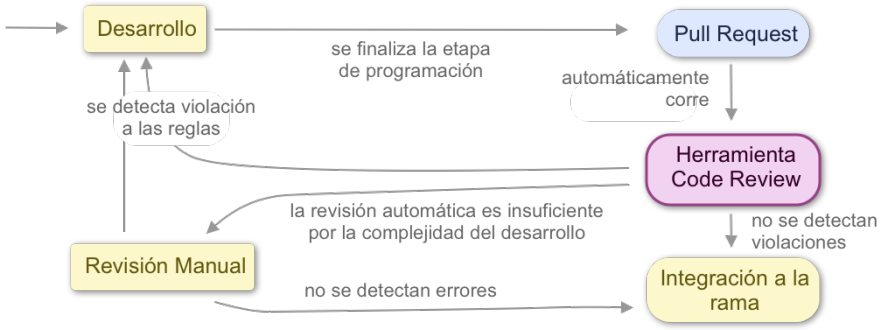

Figura 1. Diagrama con la arquitectura planteada

3.1. Modelo de Revisión de Código Basado en LLM

El sistema complementa la revisión manual mediante la detección temprana de problemas sistemáticos y errores humanos. En este sentido, el LLM actúa

como un primer filtro capaz de identificar patrones de riesgo, inconsistencias o desviaciones respecto de las buenas prácticas establecidas por el equipo.

El análisis se estructura a partir de prompts que encapsulan reglas de validación, permitiendo evaluar el código en función de criterios técnicos específicos. Este enfoque habilita la externalización del conocimiento del dominio hacia configuraciones declarativas, reduciendo la dependencia de revisores individuales y favoreciendo la estandarización del proceso.

Entre los principales ejes de análisis se incluyen la detección de credenciales expuestas, el uso incorrecto de configuraciones de entorno, la ejecución injustificada de operaciones de alto costo computacional, la gestión inadecuada de dependencias externas y la identificación de oportunidades de mejora tecnológica. Estas verificaciones, que en revisiones manuales suelen requerir múltiples iteraciones o conocimiento especializado, pueden ser abordadas de forma sistemática por el modelo.

Como ejemplo, dado un fragmento de código ingresado a través de un *pull request*, el sistema construye un contexto combinando el *diff* con los prompts definidos y delega el análisis al modelo de lenguaje. La salida generada consiste en un conjunto acotado de observaciones estructuradas, donde se reportan exclusivamente las violaciones detectadas junto con sugerencias de corrección, de acuerdo con las reglas definidas por el equipo de trabajo. Este comportamiento resulta relevante desde el punto de vista operativo, ya que minimiza el ruido en la revisión y prioriza únicamente los hallazgos accionables, alineándose con prácticas de revisión eficientes en entornos colaborativos.

3.2. Integración en el Flujo CI/CD

La herramienta se integra en la etapa de *pull request*, antes de la revisión manual, lo que permite interceptar errores frecuentes de manera temprana. Desde una perspectiva de flujo de trabajo, la automatización introduce un mecanismo de retroalimentación inmediata que reduce los tiempos de espera entre iteraciones y desacopla parcialmente la revisión de la disponibilidad de otros desarrolladores. Esto contribuye a mantener la continuidad del proceso de integración y a reducir uno de los principales cuellos de botella en equipos ágiles.

Asimismo, al delegar en el modelo la detección de problemas recurrentes, se libera capacidad cognitiva del equipo, permitiendo enfocar la revisión manual en aspectos más complejos, como decisiones de diseño o impacto arquitectónico.

3.3. Beneficios

La incorporación de un revisor automatizado basado en LLM impacta tanto a nivel técnico como organizacional. En términos de calidad, permite establecer una validación inicial consistente que reduce la propagación de errores. Desde el punto de vista operativo, optimiza el uso del tiempo al disminuir la carga asociada a tareas repetitivas y acelerar los ciclos de revisión.

Adicionalmente, el enfoque basado en reglas configurables promueve la adopción sostenida de buenas prácticas, al aplicar de forma uniforme criterios definidos por el proyecto. Esto no solo mejora la consistencia del código, sino que también contribuye a la construcción de procesos más predecibles y escalables dentro del desarrollo de software moderno.

4. Implementación

La herramienta propone la automatización parcial del proceso de revisión de código mediante modelos de lenguaje (LLMs), integrándose directamente en el flujo de desarrollo a través de GitHub. Su ejecución se activa ante la creación o actualización de un *pull request*, evento que dispara un workflow en GitHub Actions donde se inicia el análisis automático.

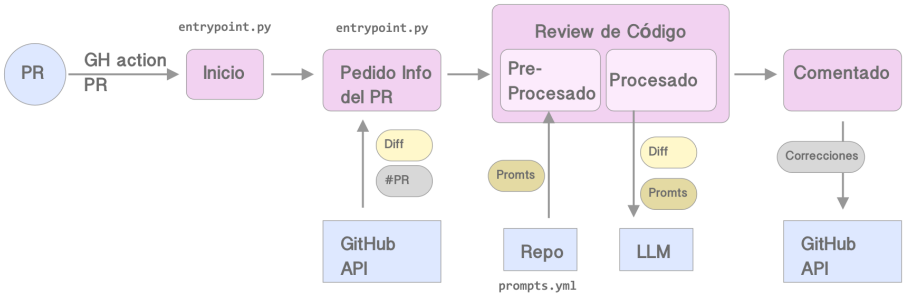

Figura 2. Diagrama de la implementación de la herramienta

El proceso comienza con la ejecución de un archivo principal (`entrypoint.py`) dentro de un contenedor, que instancia una interfaz para comunicarse con la API de GitHub y obtener el *diff* del código. Este *diff*, que representa las modificaciones propuestas, constituye la entrada principal del sistema.

4.1. Arquitectura del Sistema

La arquitectura se organiza en tres componentes: GitHub Actions, el revisor automatizado (Reviewer) y la interfaz con GitHub. GitHub Actions actúa como mecanismo de orquestación, permitiendo ejecutar el análisis de forma automática dentro del flujo CI/CD sin requerir intervención manual.

El **revisor automatizado** constituye el núcleo del sistema, ya que concentra la lógica de análisis del código. Su objetivo es evaluar los cambios introducidos en un *pull request* y generar observaciones estructuradas a partir de un conjunto de reglas definidas mediante prompts. Respecto a los parámetros de configuración

del endpoint con el LLM se experimentó con varios, siendo la *temperatura* el único que permitió mejorar la consistencia de la respuesta del modelo, teniendo varianza únicamente en los casos en los que más de 1 salida tuviera exactamente el mismo ranking.

El funcionamiento del **Reviewer** se basa en un flujo de tres etapas estrechamente integradas. En primer lugar, durante el preprocesamiento, se cargan los prompts definidos en un archivo `prompts.yml`, agrupándolos según distintos niveles de análisis (por ejemplo, línea, archivo o proyecto). Esta estructuración permite adaptar el alcance del análisis según la complejidad de cada regla.

En la etapa de análisis, el *diff* del código se combina con los prompts preprocesados para construir un contexto completo que es enviado al modelo de lenguaje. El LLM interpreta este contexto y genera un reporte que identifica posibles errores, inconsistencias o mejoras. A diferencia de un enfoque puramente descriptivo, el sistema no solo detecta problemas, sino que también propone correcciones, posicionando al modelo como un asistente activo dentro del proceso de desarrollo.

Finalmente, en el postprocesamiento, la salida del modelo se transforma en un formato estructurado (JSON), asegurando su correcta integración con GitHub y facilitando su posterior visualización como comentarios dentro del *pull request*.

Desde el punto de vista del diseño, el Reviewer se implementa a partir de una clase abstracta extensible, lo que permite desacoplar la lógica de revisión del proveedor específico del modelo de lenguaje. Esta decisión arquitectónica resulta clave para la escalabilidad del sistema, ya que posibilita incorporar nuevos modelos o estrategias de procesamiento sin modificar la estructura general.

4.2. Diseño de Prompts

El diseño de los prompts constituye el mecanismo central mediante el cual se controla el comportamiento del modelo. En lugar de depender de entrenamiento adicional, el sistema utiliza prompts estructurados que definen explícitamente el contexto del modelo, el formato de entrada, los objetivos del análisis y las condiciones de respuesta.

En este contexto, los prompts de tipo base establecen el marco general de la tarea, mientras que los prompts de tipo regla introducen validaciones específicas sobre el código. Estas reglas no solo permiten detectar errores, sino también clasificarlos y asociarlos a una corrección sugerida, generando así una salida más útil para el desarrollador. Dentro de estas estructuras de prompts vemos implementadas diferentes técnicas de prompt engineering como son el *role assignment*, por ejemplo: *"Eres un experto Data Engineer con conocimientos en Python y SQL, tu tarea es hacer code review de un pull request en GitHub. Se te proporcionará el código en Python y SQL."* que da contexto sobre la función que cumple el LLM dentro de la tarea. Luego y también dentro del prompt base está la aplicación del principio de *Structured Chain of Though* que expone de una manera cuasi programática los pasos a seguir:

1. Analizar cada línea de código.
2. Señalar exclusivamente aquellas líneas que violen 1 o más reglas.

3. Llenar el campo corrección sugerida con una correción sugerida.

4. Volver al punto 1 si existe otra línea.

5. Llenar el ${filename} con el nombre del archivo donde se encontraron los errores.

Adicionalmente, los ejemplos de salida sirve para ajustar el estilo de la respuesta. El siguiente prompt ejemplifica una salida para una regla sobre claves expuestas:

```
## Regla 1: Claves Expuestas
**corrección sugerida:** Usar variables de entorno para manejo de claves.
**línea de código:** Número y línea de código que contiene la clave
    expuesta donde se lea el string expuesto.
```

Por último y también dentro de un prompt de tipo regla vemos un ejemplo de un prompt negativo, que pone de manifiesto cuál es un resultado no deseado: *Si no existen claves expuestas no devuelvas errores de este tipo.*

Estas estrategias permiten reducir la ambigüedad en la interpretación del código y mejorar la consistencia de las respuestas generadas.

4.3. Interacción con GitHub

La clase `GitHubInterface` encapsula la interacción con la API de GitHub, permitiendo obtener el *diff* del *pull request* y publicar automáticamente los resultados del análisis como comentarios. De este modo, el sistema se integra de manera transparente en el entorno de trabajo del desarrollador, sin requerir herramientas adicionales.

La ejecución de la herramienta requiere la configuración de un workflow, un archivo de prompts y credenciales de acceso tanto al modelo de lenguaje como a la API de GitHub. Una vez configurado, el sistema produce como salida un análisis automatizado del código que se publica directamente en el *pull request*.

La solución desarrollada permite automatizar parcialmente el proceso de revisión de código mediante un enfoque basado en prompts y consumo de modelos vía API. El énfasis en el diseño del Reviewer y en la estructuración de los prompts posibilita construir una herramienta flexible, escalable y fácilmente integrable en flujos reales de desarrollo, reduciendo la carga de trabajo sin reemplazar completamente la revisión humana.

5. Evaluación

La herramienta fue evaluada en tres dimensiones que resultan complementarias: carga, aceptación y factibilidad. Estas pruebas permiten validar no solo su desempeño técnico, sino también su utilidad práctica y viabilidad de adopción en entornos reales de desarrollo.

5.1. Prueba de Carga

El objetivo de esta prueba fue determinar cuánto código puede procesar la herramienta sin comprometer el rendimiento ni la calidad de los resultados. La hipótesis inicial planteó que, al aumentar la cantidad de líneas analizadas, podrían surgir limitaciones tanto en los recursos computacionales disponibles como en la capacidad del LLM para mantener respuestas consistentes.

Para la evaluación seleccionamos 6 PRs reales, clasificados según su volumen: PR pequeños (menos de 20 LOC), medianos (menos de 100 LOC) y PR grandes (+100 LOC). El procedimiento consistió en (1) ejecutar la herramienta sobre cada PR (2) medir el tiempo de ejecución y tamaño de los prompts y (3) registrar los resultados en un gráfico de dispersión. Analizamos si existía una relación entre el tamaño del PR y el tiempo de respuesta, buscando identificar la eventual necesidad de fragmentar entradas de gran volumen antes de su procesamiento.

No observamos una correlación entre la cantidad de LOC y el tiempo de respuesta. Además verificamos que la herramienta puede procesar PR de gran tamaño (2000+ LOC) sin degradaciones significativas. Incluso en los casos de mayor tamaño, el tiempo de respuesta se mantuvo por debajo de los 40 segundos, valor aceptable para el flujo de revisión de código. Como limitación, en proyectos compuestos principalmente por PR pequeños, el beneficio relativo podría verse atenuado por el costo fijo inicial asociado a cada ejecución. Del otro lado del espectro, respecto a los PR de mayor tamaño, las herramientas actuales soportan ventanas de contexto de hasta 1m de tokens, que consideramos excede la escala de un PR promedio, por lo cual no se realizan actualmente validaciones de tamaño para este caso (si bien no se descarta para el futuro).

5.2. Prueba de Aceptación

Esta prueba buscó determinar si los comentarios generados por la herramienta resultaban útiles y relevantes para desarrolladores profesionales. Se realizó una encuesta a participantes junior, semi-senior y senior, presentándoles 6 comentarios producidos por la herramienta a partir de los casos utilizados en la prueba de carga. Cada comentario fue evaluado bajo la premisa *"La revisión es de buena calidad"*, mediante una escala de 0 (revisión deficiente) a 5 (revisión excelente). Se consultó también si incorporarían la herramienta a su flujo de trabajo.

El promedio global obtenido fue de **3,46** sobre 5. Los escenarios mejor puntuados fueron *Migración de lenguaje deprecado* y *Uso de operación con alto costo computacional*, ambos con un promedio de **3,75**. El caso con menor valoración obtuvo **3,00**, asociado a comentarios percibidos como extensos o redundantes. Respecto de la adopción, el **75 %** de los participantes indicó que incorporaría la herramienta, mientras que el **25 %** respondió negativamente.

Los resultados sugieren una percepción positiva y una aceptación mayoritaria. Si bien hay margen de mejora, la propuesta muestra valor práctico y potencial de adopción en entornos reales de desarrollo.

5.3. Prueba de Factibilidad

La tercera dimensión de evaluación analizó la factibilidad de adopción de la herramienta desde una perspectiva económica. Se realizó un análisis basado en *Total Cost of Ownership* (TCO), considerando el costo de procesamiento de tokens, volumen de código analizado y frecuencia de uso. Se utilizó como caso de estudio el repositorio de una empresa argentina de logística, empleando el modelo *OpenAI ChatGPT-4o (2024-02-15 preview)*.

En una primera evaluación, con 97 líneas de entrada y 1614 tokens de entrada, el costo estimado fue de **4,985 ARS**. En una segunda evaluación, sobre 2720 líneas de entrada y 47 718 tokens, el costo ascendió a **131,033 ARS**.

Los resultados muestran que, en PRs pequeños, gran parte del costo viene del prompt de contexto fijo, mientras que en cambios de mayor tamaño predomina el costo asociado al código analizado. En consecuencia, la herramienta resulta relativamente más conveniente en revisiones medianas o grandes que en cambios atómicos. Asimismo, cada nueva ejecución sobre un mismo PR incrementa nuevamente el costo total, ya que reprocesa la entrada completa. Por ello, la relación costo-beneficio mejora en equipos con revisiones de mayor volumen o donde la detección temprana de errores evita retrabajo posterior.

En síntesis, los costos observados resultan bajos en términos absolutos y permiten estimar la adopción en otros contextos, utilizando la conversión entre líneas de código, tokens procesados y frecuencia esperada de uso.

6. Conclusión

Los resultados de las pruebas confirman que la herramienta es viable técnica, operativa y económicamente. En términos de desempeño, demostró ser capaz de procesar *pull requests* de distintos tamaños sin mayores degradaciones, mostrando escalabilidad dentro de entornos reales de desarrollo. Asimismo, las evaluaciones de aceptación reflejan una predisposición positiva por parte de los desarrolladores, quienes valoraron especialmente la rapidez en la generación de observaciones preliminares y su utilidad como apoyo en la revisión manual. Desde el punto de vista económico, el costo de operación basado en consumo de APIs fue bajo en términos absolutos, consolidando la factibilidad del enfoque.

Estos resultados se alinean directamente con la propuesta inicial del trabajo, que planteaba la posibilidad de automatizar parcialmente el code review sin necesidad de entrenamiento adicional de modelos. La evidencia empírica demuestra que, mediante técnicas de prompt engineering y una adecuada integración en pipelines de CI/CD, los LLMs pueden actuar como un primer filtro preliminar, detectando errores frecuentes y malas prácticas. De este modo, la herramienta cumple su objetivo como sistema de asistencia que mejora la eficiencia del proceso sin reemplazar el criterio humano, en línea con un enfoque de inteligencia artificial centrada en el usuario.

En cuanto al trabajo futuro, se identifican oportunidades para ampliar las capacidades del sistema mediante la incorporación de mayor contexto, como documentación del proyecto, arquitectura o historial de cambios, lo que permitiría

mejorar la calidad de las recomendaciones. También se plantea la exploración de enfoques más avanzados, como arquitecturas multiagente especializadas y técnicas de Retrieval-Augmented Generation (RAG), así como la evaluación del desempeño en otros lenguajes y dominios de desarrollo. Estas líneas apuntan a fortalecer la precisión del análisis y a expandir la aplicabilidad de la herramienta en distintos escenarios de ingeniería de software.

Referencias

Bosu, A., & Carver, J. C. (2013). Impact of peer code review on peer impression formation: A survey. *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*.

Bosu, A., Greiler, M., & Bird, C. (2015). Characteristics of Useful Code Reviews: An Empirical Study at Microsoft. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 146-156. https://doi.org/10.1109/MSR.2015.21

Frömmgen, A., Austin, J., Choy, P., Ghelani, N., Kharatyan, L., Surita, G., Khrapko, E., Lamblin, P., Manzagol, P.-A., Revaj, M., Tabachnyk, M., Tarlow, D., Villela, K., Zheng, D., Chandra, S., & Maniatis, P. (2024). Resolving Code Review Comments with Machine Learning.

Jason Wei and Xuezhi Wang and Dale Schuurmans and Maarten Bosma and Brian Ichter and Fei Xia and Ed H. Chi and Quoc V. Le and Denny Zhou. (2023). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.

Jia Li and Ge Li and Yongmin Li and Zhi Jin. (2023). Structured Chain-of-Thought Prompting for Code Generation.

Junyi Lu and Lei Yu and Xiaojia Li and Li Yang and Chun Zuo. (2023). LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning.

Li, Z., Lu, S., Guo, D., Duan, N., Jannu, S., Jenks, G., Majumder, D., Green, J., Svyatkovskiy, A., Fu, S., & Sundaresan, N. (2022). Automating Code Review Activities by Large-Scale Pre-training.

Licorish, S. A., Holvitie, J., Hyrynsalmi, S., Leppänen, V., MacDonell, S. G., & Buchan, J. (2016). Adoption and Suitability of Software Development Methods and Practices.

Pornprasit, C., & Tantithamthavorn, C. (2024). Fine-tuning and prompt engineering for large language models-based code review automation. *Inf. Softw. Technol.*, *175*(107523), 107523.

Rasheed, Z., Sami, M. A., Waseem, M., Kemell, K.-K., Wang, X., Nguyen, A., Systä, K., & Abrahamsson, P. (2025). AI-powered Code Review with LLMs: Early Results. https://arxiv.org/abs/2404.18496

Shneiderman, B. (2022). Human-centered AI: ensuring human control while increasing automation. *Proceedings of the 5th Workshop on Human Factors in Hypertext*. https://doi.org/10.1145/3538882.3542790

TrustRadius. (2022). Agile Project Management Software Survey.