QoCode: Evaluación automática de calidad de código en tareas de programación

Victoria A. Varani[1](0009-0000-3493-8783), Bruno Ventrici[1](0009-0009-9720-4061), Gonzalo Pablo Fernández[1](0009-0001-6146-8462) y Christian Cossio-Mercado[1,2](0000-0002-2535-3078)

[1] Universidad de Buenos Aires. Facultad de Ciencias Exactas y Naturales. Departamento de Computación. Buenos Aires, Argentina. {vvarani,bventrici,gpfernandez,ccossio}@dc.uba.ar

[2] CONICET-Universidad de Buenos Aires. Instituto de Ciencias de la Computación (ICC). Buenos Aires, Argentina.

Resumen En los últimos años, el aumento sostenido de la matrícula en carreras de informática impuso desafíos crecientes a los equipos docentes, en particular en la evaluación de tareas de programación. Como respuesta a este problema, se comenzaron a utilizar más ampliamente herramientas de corrección automática de código. Sin embargo, estas se concentran casi exclusivamente en la corrección funcional, generalmente a través de conjuntos más o menos extensos de casos de prueba. En consecuencia, faltan herramientas que generen retroalimentación estructurada con enfoque pedagógico en dimensiones de la calidad de código como la legibilidad y la mantenibilidad. En este trabajo se presenta QoCode, una herramienta web de análisis estático de código orientada a generar retroalimentación automática y pedagógicamente estructurada sobre código Python. QoCode integra validaciones sobre legibilidad y mantenibilidad propias, basadas en el árbol de sintaxis abstracta, y se complementa con verificaciones de linters reconocidos, como Pylint y Flake8. Además, QoCode permite al docente configurar explícitamente qué aspectos evaluar, de acuerdo con las necesidades de su curso y sus estudiantes. Finalmente, se incluye una evaluación preliminar sobre entregas de código de dos cursos introductorios universitarios. Los resultados sugieren que QoCode detecta defectos de manera consistente y escalable, complementando la visión docente en aspectos que habitualmente quedan fuera de la retroalimentación automática, como el uso de números mágicos, nombres demasiado cortos y funciones poco legibles.

Palabras claves: calidad de código, feedback automático, enseñanza de programación, análisis estático, Python

QoCode: Automated Code Quality Assessment in Programming Assignments

Abstract. In recent years, the sustained growth in enrollment in computer science degree programs has posed increasing challenges for instructors, particularly with regard to the assessment of programming assignments. As a result, the use of automated code assessment tools has become more widespread. However, these tools focus almost exclusively on functional correctness, typically evaluated through sets of test cases, overlooking essential aspects such as code readability and maintainability. Thus, there is still a lack of tools that provide structured, pedagogically oriented feedback on code quality dimensions such as readability and maintainability. This paper presents QoCode, a web-based static code analysis tool aimed at generating automated, pedagogically structured feedback on Python code. QoCode incorporates custom analyses of readability and maintainability based on the abstract syntax tree, along with checks from well-known linters such as Pylint and Flake8. In addition, it allows instructors to explicitly configure which aspects to assess, according to course-specific requirements. Finally, the paper includes a preliminary evaluation based

on code submissions from two introductory courses at an Argentine public university. The results show that QoCode consistently and scalably detects defects, complementing instructors' assessments on aspects that are typically left out of automated feedback, such as the use of magic numbers, overly short names, and functions with poor readability.

Keywords: code quality, automatic feedback, teaching of programming, static analysis, Python

1 Introducción

Las carreras universitarias relacionadas a la informática han experimentado un crecimiento sostenido en su matrícula tanto a nivel internacional (Camp et al., 2017) como en Argentina. Este crecimiento impacta directamente en los equipos docentes, que deben trabajar con cursos cada vez más numerosos sin sacrificar la calidad del seguimiento y retroalimentación (en inglés, *feedback*) ofrecidos a los estudiantes.

En las materias introductorias de programación, los docentes suelen recurrir a conjuntos de casos de prueba y herramientas de evaluación automática basadas en tests unitarios para poder atender a grandes cantidades de estudiantes (Messer et al., 2024). Sin embargo, este enfoque evalúa exclusivamente la corrección funcional, dejando de lado la calidad de código. Esta omisión puede tener consecuencias serias: los estudiantes consolidan hábitos de programación deficientes que luego resultan difíciles de corregir (Effenberger & Pelánek, 2022; Keuning et al., 2017). Un código de baja calidad puede derivar en mayores costos de mantenimiento, dificultades de extensión y hasta vulnerabilidades de seguridad (Keuning et al., 2017). En el ámbito educativo, la falta de retroalimentación específica sobre calidad implica que los estudiantes no disponen de criterios para evaluar su propio trabajo (Ala-Mutka & Jarvinen, 2004), y sin retroalimentación dirigida, esos defectos tienden a mantenerse y consolidarse (Effenberger & Pelánek, 2022).

Para que las devoluciones sean formativas deben abarcar no sólo la corrección funcional, sino también la calidad del código. Sin embargo, este tipo de revisión requiere tiempo y conocimiento experto, lo que resulta difícil de sostener en cursos multitudinarios.

Para abordar esta problemática, proponemos *QoCode*: una herramienta web de análisis estático orientada a generar retroalimentación automática sobre calidad de código en contextos educativos (por ejemplo, en cursos universitarios), con foco en legibilidad y mantenibilidad. Las contribuciones principales son las siguientes:

- Una propuesta conceptual que descompone legibilidad y mantenibilidad en cuatro y cinco categorías respectivamente, cada una con un catálogo de medidas concretas, que pueden resolverse de forma automática.
- El diseño e implementación de *QoCode*, que integra *linters* existentes con validaciones propias sobre el árbol sintáctico abstracto (con sus siglas en inglés, AST), y produce retroalimentación en lenguaje natural pedagógicamente orientado.
- Una evaluación preliminar sobre entregas reales de dos cursos introductorios de la Facultad de Ciencias Exactas y Naturales (FCEN) de la Universidad de Buenos Aires (UBA), contrastando hallazgos automáticos con devoluciones docentes.

2 Estado del Arte

Messer et al. (2024) realizaron una revisión sistemática de 121 artículos publicados entre 2017 y 2021 sobre herramientas de retroalimentación automática en la enseñanza de la programación.

En su trabajo proponen descomponer la calidad de código en cuatro aspectos: **corrección**, **mantenibilidad**, **legibilidad** y **documentación**. Su análisis muestra un predominio abrumador del *feedback* sobre corrección funcional, mientras que la investigación sobre legibilidad y mantenibilidad es señalada como 'incipiente'. Además, señalan que el *feedback* de muchas herramientas automáticas resulta pedagógicamente inadecuado: suele ser poco claro, muy técnico y no orienta sobre cómo mejorar el código (Alberola & García-Fornes, 2014; Keuning et al., 2017, 2018).

Stegeman et al. (2016) abordan el diseño sistemático de rúbricas para cursos introductorios frente al doble problema de la escalabilidad del proceso de corrección y la consistencia del *feedback* entre evaluadores. Así, proponen una rúbrica que jerarquiza los errores según su gravedad, evitando evaluaciones planas donde todos los defectos reciben el mismo peso. Este tipo de *feedback* permite comunicar no solo qué problemas requieren corrección, sino por qué ciertos problemas son más relevantes que otros. Entre las características que identifican, varias se relacionan directamente con legibilidad: *nombres*, *disposición*, *formato* y *expresiones*.

Effenberger y Pelánek (2022) propusieron un catálogo inicial de 32 defectos de calidad de código que Řechtáčková et al. (2025) ampliaron a más de 100. Los mismos autores desarrollaron *EduLint* (Řechtáčková et al., 2025), una herramienta de análisis de código *Python* para estudiantes iniciales, que integra *linters* existentes con más de 70 detectores propios. Un *linter* es una herramienta de análisis estático que revisa el código fuente sin ejecutarlo, reportando posibles errores, inconsistencias y problemas de estilo, indicando las líneas de código involucradas.

Sin embargo, ninguna de las herramientas mencionadas aborda de manera integrada la generación de retroalimentación pedagógicamente estructurada sobre legibilidad y mantenibilidad en contextos educativos, lo que constituye el principal vacío que este trabajo busca cubrir.

3 Dimensiones de calidad de código

QoCode hace foco en dos dimensiones de la calidad: legibilidad y mantenibilidad. La legibilidad no se limita a la claridad sintáctica, sino que también incluye la comprensión semántica derivable de los nombres, las expresiones y el formato. La mantenibilidad abarca la construcción del código, buscando que tenga un diseño y complejidad adecuados. En este artículo no se aborda el aspecto de documentación del código, que se deja como trabajo futuro.

3.1 Legibilidad

Se proponen cuatro características que organizan los problemas más frecuentes en trabajos de estudiantes iniciales. Esta descomposición extiende los antecedentes de Stegeman et al. (2016) y Řechtáčková et al. (2025), cuyos aspectos de legibilidad aparecen agrupados sin una caracterización detallada.

Claridad semántica de los nombres Abarca los problemas relacionados con la semántica de los identificadores: variables, parámetros, procedimientos y funciones. Un nombre semánticamente claro permite al lector inferir la intención del código sin entrar en la implementación; un nombre ambiguo o engañoso dificulta la comprensión del programa.

Esta característica incluye medidas como *nombres engañosos por cardinalidad*, que sugieren un tipo distinto al real; *nombres engañosos por significado*, que implican una acción diferente; *nombres de un solo carácter fuera de contexto*, usados fuera de ciclos donde resultan ambiguos; *nombres sin vocales o sin sentido*; *gramática inconsistente* entre elementos equivalentes; y *nombres incompletos*, demasiado genéricos.

Estilismo de nombres Comprende lo relativo al formato y la coherencia visual de los identificadores. Como el estilismo visual busca coherencia mediante la combinación uniforme de elementos, aquí se busca que los nombres sigan lineamientos uniformes que faciliten recorrer el código. Abarca diferentes medidas: *nombres excesivamente largos*, que superan un umbral de longitud predefinido y dificultan la lectura en expresiones complejas o estructuras anidadas; *errores ortográficos*, que generan confusión y desconfianza; *uso inconsistente de casing*, como mezclar *camelCase* con *snake_case* dentro del mismo archivo; y *uso de palabras reservadas* como nombres de identificadores, que ocultan la funcionalidad original del lenguaje y pueden causar errores difíciles de detectar.

Formato del código Abarca la estructura visual y la consistencia del código. El objetivo es que fragmentos similares presenten un formato homogéneo que permita identificar rápidamente los bloques lógicos. Incluye varias medidas: *indentación y espaciado incorrecto o inconsistente*, que rompe la uniformidad visual y aumenta la carga cognitiva; *declaración de variables lejos de su uso*, que obliga al lector a retener información innecesariamente; y *múltiples sentencias en una misma línea*, que dificultan la lectura y la depuración (en inglés, *debugging*).

Claridad semántica de los valores Aborda la pérdida de información semántica asociada a los valores que maneja el programa. Es deseable que quien lee el código pueda inferir qué representa un valor y que ese significado sea consistente a lo largo del programa. Incluye el *uso de magic numbers*, constantes numéricas sin nombre usadas directamente en comparaciones, y la *reasignación de variables con cambio de tipo o propósito*, que hace perder la referencia semántica del valor que recuerda la variable en cada momento del programa.

3.2 Mantenibilidad

La mantenibilidad refiere a qué tan bien el estudiante ha implementado el código, en términos de diseño y complejidad (Messer et al., 2024). Estas características aportan a que el código sea fácil de mantener en el tiempo (tanto para realizar correcciones como modificaciones) y que las funciones y procedimientos que se definen puedan ser reutilizados en contextos similares. Se adoptan cinco categorías de defectos basadas en el trabajo de Řechtáčková et al. (2025):

Código repetido Agrupa los defectos relacionados con la repetición de código. Así, al evitar la repetición se previenen inconsistencias al corregir errores en la lógica, además de facilitar la modificación del programa ante un cambio de requerimientos. Incluye *expresiones duplicadas*, cuando una misma expresión compleja se repite en lugar de abstraerse en una variable o función; *secuencias de comandos duplicadas*, cuando un grupo de instrucciones similares podría reemplazarse por una repetición o encapsularse en un procedimiento; *bloques de código duplicados*, que podrían parametrizarse en un único procedimiento; *ramas secuenciales de una alternativa (*if*) con el mismo cuerpo*, que podrían unificarse mediante una disyunción; y *ramas de una alternativa con código extraíble*, cuando el inicio o el final de cada rama es idéntico y podría moverse fuera del bloque.

Diseño pobre Aborda el código innecesariamente complejo debido a elecciones de diseño deficientes, ya que un código bien diseñado facilita su lectura y comprensión. Cualquier mejora en esta categoría requiere modificar el código de forma no trivial, sin alterar su comportamiento. Incluye *funciones con múltiples responsabilidades o demasiadas líneas*, que deberían descomponerse en unidades más pequeñas y cohesivas; *uso inadecuado de estructuras de datos*, como

usar una lista para guardar un resultado intermedio cuando bastaría un número; y *variables dependientes innecesarias*, cuando una puede calcularse a partir de la otra en lugar de mantenerse sincronizadas manualmente.

Simplificable Agrupa el código que puede simplificarse con cambios locales sin introducir nuevas construcciones. El objetivo es reducir la complejidad del flujo de control para favorecer la comprensión del programa y facilitar la detección de errores. Incluye *ramas redundantes en alternativas*, *expresiones booleanas simplificables*, como comparaciones con constantes booleanas en lugar de usar directamente la expresión (a veces conocido informalmente como el "síndrome del miedo al booleano"), *alternativas anidadas* y *variables intermedias redundantes* cuando el valor puede usarse directamente en la expresión que la utiliza.

Construcción inadecuada Se centra en detectar código que resultaría más explícito y simple usando una construcción diferente del lenguaje. Una construcción adecuada comunica de forma explícita la intención del programador, reduciendo la distancia entre el código y el problema que resuelve. Incluye usar *for* en lugar de *while* cuando el número de iteraciones es conocido de antemano, usar *in* en lugar de comparaciones múltiples por disyunción y usar *with* en lugar de *open* y *close* para el manejo de archivos.

Código sin efecto Agrupa el código que no afecta el resultado de la ejecución. Su presencia suele deberse a lógica obsoleta o mal diseñada, y produce ruido innecesario que dificulta la comprensión del programa. Todos los defectos de esta categoría se resuelven eliminando código, sin necesidad de reordenarlo ni de incorporar nuevas construcciones. Incluye *variables asignadas, pero nunca utilizadas*, *módulos importados sin usar*, *código inalcanzable*, *código comentado* olvidado en la versión final y *uso de impresiones para depuración* que generan salida no intencional en el programa entregado.

4 Diseño e Implementación de QoCode

4.1 Descripción general

QoCode[1] es una herramienta web de análisis automático orientada a brindar *feedback* sobre la calidad de código en contextos educativos. Recibe como entrada el código fuente de la entrega de un estudiante y permite al docente seleccionar explícitamente qué dimensiones y características desea evaluar. La configuración busca adaptar el análisis a cada contexto, evitando devoluciones indiscriminadas que no resulten formativas. El objetivo no es únicamente detectar problemas, sino acompañar el proceso de aprendizaje, promoviendo buenas prácticas desde las etapas iniciales de la formación.

A diferencia de los *linters* tradicionales y otras herramientas, que presentan salidas en formato crudo orientadas a desarrolladores experimentados, *QoCode* transforma los resultados en mensajes estructurados y pedagógicos. Cada hallazgo incluye la línea de código afectada, una descripción del problema en lenguaje accesible, y una recomendación concreta para su resolución. De esta forma, las sugerencias formulan ejemplos de buenas prácticas sin reemplazar el proceso de diseño ni la toma de decisiones propias del estudiante.

También se diferencia de *EduLint* en cuanto al enfoque pedagógico. *EduLint* presenta todos los defectos detectados de manera individual y exhaustiva, lo que puede inducir a los estudiantes a corregir errores mecánicamente al final del proceso (usando la herramienta como un filtro a superar) en lugar de desarrollar criterios de calidad desde el inicio. En contraste, *QoCode* permite al

[1] QoCode, sitio web: qocode.dc.uba.ar

docente configurar qué aspectos evaluar, presenta el *feedback* agrupado por categoría conceptual, e incluye descripciones en prosa y consejos concretos orientados al aprendizaje. Esta orientación responde a la observación de Stegeman et al. (2016) de que un *feedback* formativo debe contextualizar los defectos dentro de un marco conceptual que facilite su comprensión y promueva el aprendizaje de buenas prácticas.

4.2 Arquitectura

QoCode implementa una arquitectura de tres capas que se ejecuta completamente como un proceso local en *Python*, sin dependencias de servicios externos. La arquitectura elegida evita el uso de *frameworks* pesados, priorizando la simplicidad, la transparencia y el control sobre el funcionamiento interno del sistema:

1. **Capa de presentación:** un servidor HTTP liviano complementado con el uso de *sockets* para el manejo de múltiples *requests* concurrentes. Esta capa es genérica y está desacoplada de la lógica específica de la aplicación: responde solicitudes *GET* sirviendo el *frontend* completo y solicitudes *POST* invocando la función de análisis.
2. **Capa de aplicación:** extrae el código fuente cargado, identifica las validaciones seleccionadas por el docente, coordina la ejecución de los análisis, filtra los resultados en función de los criterios elegidos y estructura los hallazgos para la presentación en la interfaz.
3. **Motor de análisis:** ejecuta las verificaciones sobre el código. Integra *linters* externos (*Pylint* y *Flake8*) con un conjunto de validaciones propias basadas en el árbol abstracto sintáctico (AST).

Dado que los *linters* generan sus propios códigos y mensajes de error, se implementó una tabla de traducción que convierte estos resultados a un formato pedagógico unificado. Todas las validaciones, independientemente de su origen, estructuran sus hallazgos en un registro con los siguientes campos: *línea*, *mensaje*, *descripción*, *consejo*, *tipo* y *check*.

Las validaciones propias amplían el alcance más allá de las herramientas tradicionales, cubriendo aspectos como: detección de nombres excesivamente largos, uso de números mágicos, comparación de booleanos con constantes (por ejemplo, $x == True$), uso de enteros como booleanos o viceversa, reasignación de variables con cambio de tipo, duplicación de lógica entre ramas de una alternativa, y detección de código comentado.

En conjunto, esta arquitectura permite transformar un conjunto heterogéneo de herramientas y reglas en un sistema coherente, capaz de ofrecer retroalimentación clara, estructurada y pedagógicamente útil.

4.3 Interfaz de usuario

La interfaz sigue un flujo secuencial de cinco etapas que guían al usuario a lo largo del proceso de análisis. La disposición lineal responde a la lógica natural de la tarea, permitiendo avanzar de manera progresiva y sin ambigüedades.

Etapa 1 — Carga de archivos. Área central que admite selección manual, *drag-and-drop* y archivos comprimidos (*.zip*), incluyendo múltiples archivos simultáneos. La especificación explícita de los formatos soportados reduce posibles errores en la interacción.

Etapa 2 — Configuración del análisis. El docente selecciona qué dimensiones evaluar mediante casillas de verificación. La interfaz sigue un patrón de expansión progresiva: las categorías se muestran inicialmente comprimidas y pueden desplegarse para acceder a mayor detalle. A partir de pruebas de usabilidad previas, se consolidó un enfoque de selección granular para todas las dimensiones (como se observa en la Figura 1). Esta estrategia, que permite la selección individual

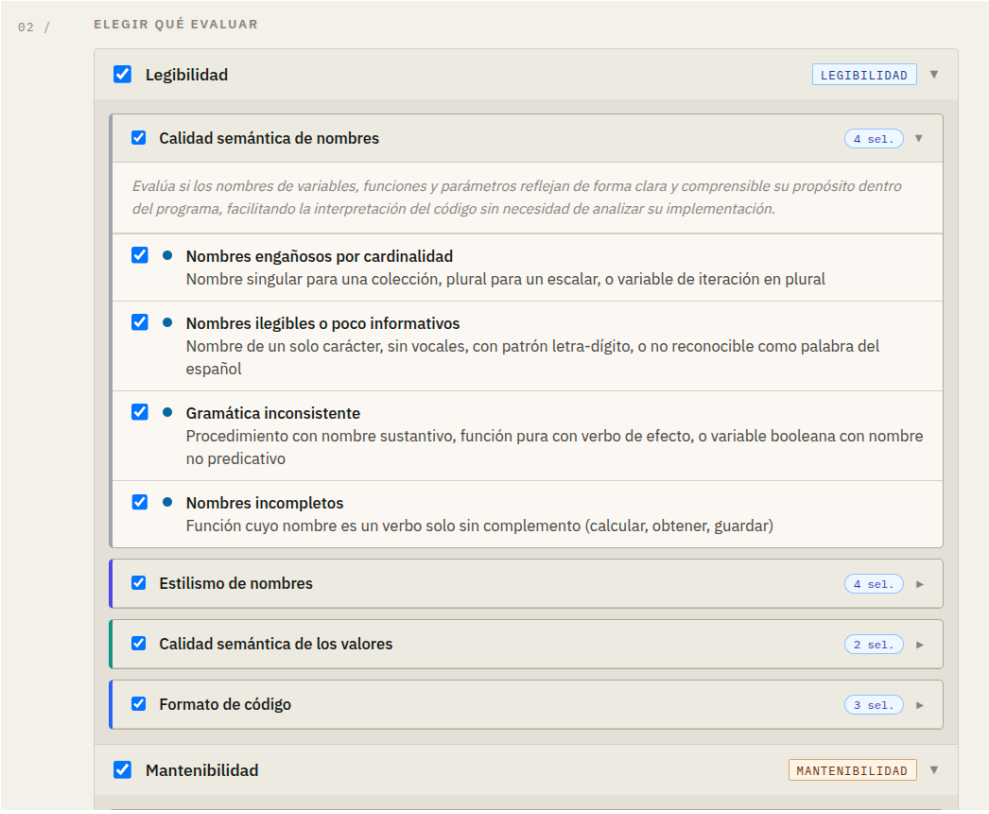

Figura 1. Configuración del análisis: se muestran los sus subgrupos que permiten la selección individual de las medidas de cada dimensión.

de cada medida en lugar de subgrupos indivisibles, demostró ser la más adecuada para garantizar la efectividad pedagógica y la flexibilidad en el contexto educativo.

Etapa 3 — Ejecución. Un botón de “Analizar Código” consolida la configuración e inicia el análisis, actuando como punto de transición entre la definición de parámetros y la obtención de resultados.

Etapa 4 — Visualización de hallazgos. Los resultados se organizan jerárquicamente retomando las categorías definidas en la configuración. Para cada dimensión se muestra un resumen cuantitativo de hallazgos, permitiendo una visión inmediata del estado del código. Cada grupo puede desplegarse para acceder al detalle individual: línea afectada, descripción del problema y consejo de mejora, como muestra la Figura 2. El uso de colores e indicadores permite diferenciar visualmente los tipos de problemas, facilitando la identificación de patrones recurrentes.

Durante el proceso de evaluación de la plataforma, descripto en la sección 5), se analizaron entregas realizadas por estudiantes. Dicho análisis reveló el uso de variables no inicializadas. Este defecto crítico se clasificó en la categoría "Otros", y la herramienta se configuró para incluirla como una advertencia obligatoria en todos los análisis.

Etapa 5 — Descarga de reportes. La sección de visualización permite exportar informes detallados para almacenar o compartir los resultados del análisis de cada entrega. Asimismo, la herramienta ofrece la opción de descargar el análisis de la totalidad de las entregas en un único archivo. Los reportes están disponibles en formatos *texto plano* (.txt), *Markdown* (.md) y PDF.

Figura 2. Vista de hallazgos de *QoCode*: cada defecto indica la línea afectada, describe el problema en prosa accesible e incluye un consejo pedagógico concreto. El resumen cuantitativo por subcategoría permite identificar rápidamente las áreas problemáticas más frecuentes.

5 Evaluación

Una vez implementada una primera versión funcional de *QoCode*, se llevó a cabo una evaluación preliminar con el objetivo de analizar su desempeño en un contexto real. El objetivo no fue medir métricas cuantitativas exhaustivas, sino obtener evidencia cualitativa inicial. En particular, se buscó analizar la pertinencia de los hallazgos detectados y la utilidad de las sugerencias en el contexto de aprendizaje. Asimismo, esta instancia permitió identificar limitaciones, posibles mejoras y oportunidades de extensión.

La evaluación utilizó entregas de dos cursos introductorios de la FCEN-UBA correspondientes al ciclo lectivo 2025. En ambos casos se contaba con las devoluciones docentes reales, lo que permitió contrastar los hallazgos automáticos con la retroalimentación efectivamente provista en cada contexto.

Con estos datos, se compararon los defectos detectados automáticamente con los mencionados explícitamente en las devoluciones docentes, permitiendo evaluar el grado de solapamiento y complementariedad entre ambos enfoques.

5.1 Escenario A — Introducción a la Programación

El Escenario A abarcó el análisis de 9 trabajos prácticos de la materia *Introducción a la Programación* (segundo cuatrimestre de 2025), realizados en grupos de 3 personas, que consistieron en la implementación del juego Batalla Naval en *Python*. Algunos de los trabajos contaban con una entrega inicial y una reentrega, lo que permitió observar la evolución del código durante el proceso de corrección y analizar si los defectos señalados por *QoCode* en la primera entrega persistían o se corregían en la versión final.

Este escenario constituye el estándar de referencia más exigente para evaluar *QoCode*, porque permite contrastar sus hallazgos con retroalimentación docente de alta calidad, granular y fundamentado conceptualmente. Esto hace posible identificar con precisión tanto los aciertos de la herramienta como sus limitaciones actuales.

5.2 Escenario B — Pensamiento Computacional

El Escenario B incluyó 20 trabajos prácticos aprobados de la materia *Pensamiento Computacional* (primer cuatrimestre de 2025), realizados en grupos de 2 personas, correspondientes a cuatro temas: Scrabble, Rayuela, Blackjack y 2048. Se eligieron trabajos aprobados que contaban con al menos un comentario docente. En este contexto, el *feedback* docente se orienta predominantemente a la corrección funcional: errores lógicos, uso de herramientas no vistas en clase e interpretación de resultados. *QoCode* fue ejecutado analizando aspectos de mantenibilidad y legibilidad en cada uno de los 20 trabajos, y sus hallazgos fueron contrastados con las devoluciones reales disponibles.

6 Resultados

El Cuadro 1 resume el porcentaje de trabajos en los que *QoCode* detectó cada defecto en cada escenario, junto con el porcentaje de trabajos en los que cada defecto fue mencionado en las devoluciones docentes del Escenario A ($n_A = 9$ trabajos, $n_B = 20$ trabajos).

En el Escenario A, los defectos con mayor frecuencia de detección automática fueron el uso de números mágicos (7 trabajos de 9), los módulos importados sin usar (6 trabajos) y las expresiones repetidas (6 trabajos), ninguno de los cuales fue mencionado por los docentes en sus devoluciones. La herramienta detectó variables sin uso en 4 trabajos frente a sólo 1 mención docente, y nombres de un solo carácter en 5 trabajos frente a 3 menciones en las devoluciones. El único caso donde la detección docente superó a la automática fue la función difícil de leer (4 menciones de los docentes, contra 3 detecciones).

En el Escenario B, los nombres de un solo carácter se detectaron en 18 de los 20 trabajos analizados, las funciones difíciles de leer en 17, y el uso de números mágicos en 14. Las variables sin uso aparecieron en 16 trabajos y las expresiones repetidas en 11. De manera transversal a ambos escenarios, los defectos más frecuentemente detectados por *QoCode* fueron los nombres de un solo carácter, el uso de números mágicos, las funciones de baja legibilidad y las variables sin uso.

7 Discusión

7.1 Alcance y limitaciones de la detección automática

En términos generales, *QoCode* resulta eficaz para identificar mejoras estructurales basadas en reglas estáticas y análisis del *AST*, tales como módulos o variables sin uso, estructuras anidadas y problemas de formato. Sin embargo, la intervención docente aporta una capa de contexto semántico que la herramienta actualmente no puede capturar: permite identificar la reimplementación

Cuadro 1. Porcentaje de trabajos en los que *QoCode* detectó cada defecto, por escenario, y porcentaje de trabajos en los que el docente lo mencionó en el Escenario A ($n_A = 9$, $n_B = 20$).

Defecto	**Esc. B** ***QoCode***	**Esc. A** ***QoCode***	**Esc. A Docente**
Legibilidad			
Nombre de un solo carácter	90 %	56 %	33 %
Uso de número mágico	70 %	78 %	0 %
Variable reutilizada por funciones	10 %	0 %	0 %
Reasignación con cambio de tipo	5 %	0 %	0 %
Múltiples sentencias en una línea	5 %	0 %	0 %
Nombre excesivamente largo	5 %	0 %	0 %
Mantenibilidad			
La función es difícil de leer	85 %	33 %	44 %
Variable sin usar	80 %	44 %	11 %
Expresión repetida	55 %	67 %	0 %
Condición negada en una alternativa compuesta	30 %	22 %	0 %
Ramas redundantes en alternativas	20 %	11 %	0 %
Ramas secuenciales de una alternativa con mismo cuerpo	20 %	0 %	0 %
Código comentado	15 %	22 %	0 %
Módulo importado sin usar	10 %	67 %	0 %
Parámetro sin usar	10 %	0 %	0 %
Código repetido entre ramas de una alternativa	10 %	11 %	11 %
Comparar booleano con constante booleana	10 %	11 %	11 %
Alternativa anidada simplificable	5 %	22 %	0 %
Rama negativa redundante	0 %	11 %	11 %

innecesaria de funciones provistas por la cátedra, la necesidad de abstraer funciones con múltiples responsabilidades, o nombres que exponen la representación interna en lugar de abstraerse hacia el concepto del dominio.

Esta brecha es especialmente visible en la dimensión de legibilidad. El docente opera con una taxonomía semántica que *QoCode* no alcanza: señala variables acumuladoras cuyos nombres no comunican su naturaleza provisional (el patrón "resultado por ahora"), confusiones entre el tipo implicado por el nombre y el tipo real del valor, o funciones nombradas como comandos cuando en realidad son expresiones. Estos problemas requieren razonar sobre el significado del programa, lo que excede el alcance del análisis estático convencional. Por ejemplo, uno de los trabajos del Escenario A ilustra esto con claridad: *QoCode* no detecta ningún hallazgo de legibilidad en la versión reentregada por los estudiantes, pero el docente identifica múltiples problemas de nombres, más relacionados a lo semántico, mostrando que la ausencia de hallazgos no equivale a la ausencia de problemas en esta dimensión.

A partir de esta brecha, se diseñaron e implementaron cuatro nuevos detectores orientados específicamente a la semántica de los nombres, con el objetivo de reducir la distancia entre los hallazgos automáticos y las observaciones docentes. Los detectores abordan patrones como incongruencias de cardinalidad entre el nombre y el valor asignado, identificadores de bajo valor informativo, inconsistencias entre la categoría gramatical del nombre y el rol semántico que cumple en el código, y nombres de comandos y expresiones que no comunican su propósito. Todos combinan análisis del AST con procesamiento morfológico mediante spaCy, y comparten un principio de diseño común, ante la imposibilidad de determinar con certeza la presencia de un de-

fecto, prefieren el silencio al falso positivo. Su validación sistemática constituye una de las líneas prioritarias del trabajo futuro descripto en la Sección 8.

En cuanto a la mantenibilidad, la métrica de complejidad cognitiva utilizada por *QoCode* para detectar funciones difíciles de leer permitió establecer un umbral heurístico a partir del cual los docentes del Escenario A señalan una función como problemática y recomiendan su reestructuración. Sin embargo, se observaron discrepancias en ambas direcciones: funciones que superan el límite sin recibir observación docente, y funciones donde el docente sugiere cambios a pesar de un índice bajo. Esto indica que el umbral requiere ajuste contextual y que la complejidad cognitiva captura solo parcialmente la noción subjetiva de legibilidad que manejan los docentes expertos. En una dirección similar, *QoCode* detectó defectos que los docentes ignoraron en sus devoluciones, como código comentado o módulos importados sin usar, probablemente, porque no tenían efecto real en la ejecución de la solución.

Desde el punto de vista técnico, los trabajos con errores de sintaxis críticos bloqueaban el análisis al impedir el parseo del código fuente. Para mitigar esta limitación, se implementó un mecanismo de parseo parcial tolerante a errores. Cuando el módulo 'ast' estándar de Python no puede procesar un archivo, el sistema recurre a *parso*, una biblioteca de análisis sintáctico que, a diferencia de *ast*, continúa ante errores y los marca como nodos inválidos dentro del árbol. A partir de dicho árbol, el sistema descarta los bloques de nivel superior afectados, construye una versión saneada del archivo y aplica un mapa de correspondencia de líneas para que los hallazgos referencien siempre el archivo original. De este modo, la presencia de funciones sintácticamente inválidas no impide el análisis del resto del código.

7.2 El *feedback* docente y sus limitaciones contextuales

El contraste entre los dos escenarios revela una diferencia significativa en el tipo de *feedback* disponible. En el Escenario A los docentes proporcionaron devoluciones exhaustivas y conceptualmente fundamentadas, mientras que en el Escenario B los comentarios sobre calidad de código fueron escasos y de carácter general. En este último caso, abundaron las observaciones del tipo *"el código es prolijo y se puede seguir bastante bien"* o *"las soluciones son bastante complejas"*, sin señalar defectos concretos ni orientar hacia mejoras específicas.

Esta diferencia no debe interpretarse necesariamente como una limitación de los docentes del Escenario B. Una hipótesis plausible es que muchos de los defectos detectados por *QoCode* no son reconocidos como tales en todos los contextos de enseñanza: las buenas prácticas de programación no están universalmente estandarizadas y varían entre lenguajes, planteles docentes e instituciones. Lo que en un contexto se considera un defecto de legibilidad puede ser una convención aceptada en otro (por ejemplo, según los criterios del Escenario A, es preferible repetir una expresión antes que recordarla en una variable). En este sentido, calificar la ausencia de *feedback* docente sobre un defecto como un error de omisión sería apresurado.

Esto refuerza el aspecto central del diseño de *QoCode*: la posibilidad de que el docente configure explícitamente qué defectos marcar, adaptando el análisis a sus propios criterios pedagógicos. *QoCode* no pretende definir universalmente qué es calidad de código, sino ofrecer un instrumento flexible que cada equipo docente pueda ajustar. Así, en lugar de imponer un conjunto fijo de criterios, la herramienta permite adaptar el análisis a las buenas prácticas y convenciones de cada materia y plantel docente.

7.3 Valor diferencial respecto al *feedback* docente

Un resultado especialmente relevante es que *QoCode* detecta defectos incluso en trabajos con calificaciones muy altas y comentarios positivos. En particular, en el Escenario B algunos trabajos con notas elevadas presentaron más defectos detectados que otros con notas menores, lo que

ilustra que la percepción subjetiva de "código prolijo" no equivale a un código libre de problemas de calidad detectables en forma automática.

Uno de los hallazgos más ilustrativos del valor diferencial es el uso de números mágicos. En los trabajos que modelan el *Blackjack*, aparecen valores como 21, 11, 17 o 6, que se usan directamente en comparaciones sin estar asociados a constantes con nombre, y ningún docente señaló este aspecto a pesar de que dificulta significativamente la comprensión del código para un lector externo.

Otro de los hallazgos destacados fue el de los nombres de un solo carácter. Por ejemplo, en uno de los trabajos se los utiliza en 19 líneas distintas del código, un hábito que los docentes mencionan de manera inconsistente y que la detección sistemática de *QoCode* permite señalar en todos los casos.

Adicionalmente, el formato de código es otra dimensión donde la herramienta aporta valor que el *feedback* docente no cubre. Varios equipos presentan problemas de espaciado alrededor de operadores, mezcla de estilos de sangrado o múltiples sentencias en una misma línea, ninguno de los cuales fue señalado en las devoluciones analizadas.

En lo respectivo a la mantenibilidad, los hallazgos omitidos por los docentes incluyen funciones que superan las 60 líneas de extensión o que presentan cuatro o más niveles de anidamiento, umbrales que en el Escenario A resultaron consistentes con las observaciones docentes sobre reestructuración.

También se detectaron de forma sistemática defectos que requieren una inspección minuciosa para identificarse manualmente, como expresiones repetidas, condiciones negadas innecesarias y cláusulas *elif* redundantes.

En cuanto a las variables sin usar, los docentes señalaron únicamente los casos de asignación explícita, pasando por alto variables definidas en cabeceras de ciclos *for* sin uso en su cuerpo o descartadas al desempaquetar tuplas; defectos que, en una etapa inicial del aprendizaje, pueden considerarse de relevancia menor, pero que la herramienta detecta de forma consistente.

Finalmente, la reasignación de variables con cambio de tipo ilustra un caso donde *QoCode* y el docente detectan el mismo problema desde perspectivas complementarias. En uno de los trabajos, la variable *ganador* comienza siendo un entero y luego es reasignada como *string*; el docente lo formula desde el diseño de la función (*"hace varias cosas porque puede recibir varios tipos"*), mientras que la herramienta lo detecta desde la claridad semántica de la variable. Ambas perspectivas son complementarias y sugieren que la combinación de análisis automático y revisión docente es más poderosa que cada una por separado.

8 Conclusiones

En este trabajo se analizaron defectos de la calidad de código en dos dimensiones, legibilidad y mantenibilidad, descomponiéndolas en cuatro y cinco categorías respectivamente, cada una con un catálogo de medidas concretas que se pueden evaluar en forma automática.

Se implementó una primera versión de *QoCode*, una herramienta web de análisis estático orientada a generar retroalimentación automática sobre la calidad de código en contextos educativos, con foco en legibilidad y mantenibilidad. La herramienta integra *linters* existentes con validaciones propias sobre el *AST*, y permite al docente configurar explícitamente qué aspectos evaluar. La herramienta transforma los resultados en retroalimentación estructurada en lenguaje natural, con descripción del problema y un consejo pedagógico concreto para cada hallazgo.

QoCode fue utilizado para evaluar distintas entregas de trabajos prácticos de dos cursos introductorios de la FCEN-UBA, y se comparó el resultado con las correcciones realizadas por los docentes. Esta comparación permitió contrastar los hallazgos automáticos con la retroalimentación real brindada en cada contexto, sugiriendo que *QoCode* detecta defectos de manera consistente y escalable, complementando la visión docente sin reemplazarla.

El Escenario A analizado evidencia que, incluso cuando el docente proporciona devoluciones detalladas, *QoCode* identifica defectos adicionales basados en reglas estructurales. El Escenario B confirma que cuando el *feedback* se centra en la corrección funcional, la herramienta cubre un espacio significativo que de otro modo quedaría sin atender. En ambos escenarios, los defectos más frecuentes fueron los nombres de un solo carácter, el uso de números mágicos, las funciones de baja legibilidad y las variables sin uso.

9 Trabajo Futuro

Como trabajo futuro se identifican cuatro líneas prioritarias. La primera es la profundización del análisis semántico de nombres. Si bien los detectores incorporados representan un avance concreto en esta dirección, la brecha respecto a la taxonomía semántica que aplican los docentes expertos permanece abierta. Superarla requerirá incorporar técnicas más sofisticadas, posiblemente combinando análisis interprocedural, inferencia de tipos y modelos de lenguaje.

La segunda línea es la aplicación longitudinal de *QoCode* a lo largo de un curso completo como herramienta de acompañamiento: integrarla de forma activa durante la cursada permitiría estudiar si el *feedback* iterativo sobre calidad produce una mejora observable en las entregas sucesivas y si los defectos señalados tempranamente tienden a no repetirse.

La tercera es la validación del *feedback* generado mediante un panel de expertos en programación y didáctica, para identificar qué tipos de hallazgos son consistentemente valiosos, cuáles generan falsos positivos en ciertos contextos y qué aspectos del *feedback* textual resultan más comprensibles para estudiantes iniciales. Esta validación permitiría calibrar y mejorar los detectores existentes de forma fundamentada. Ambas líneas son complementarias: la aplicación en curso provee el escenario real donde medir el impacto, mientras que la validación por expertos provee el criterio de calidad para interpretar esos resultados.

Por último, la cuarta línea de trabajo futuro incluye la incorporación de niveles de severidad en los hallazgos (por ejemplo, oportunidades de mejora vs. advertencias de defectos más graves) y la extensión del soporte a otros lenguajes utilizados en cursos introductorios junto con detectores para el aspecto de documentación.

Referencias

Ala-Mutka, K., & Jarvinen, H.-M. (2004). Assessment Process for Programming Assignments. En *Proceedings of the IEEE International Conference on Advanced Learning Technologies* (pp. 181-185). IEEE Computer Society.

Alberola, J. M., & García-Fornes, A. (2014). Using Feedback for Improving the Learning Process in Programming Courses. *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, *9*(2), 49-56. https://doi.org/10.1109/RITA.2014.2317529

Camp, T., Adrion, W. R., Bizot, B., Davidson, S., Hall, M., Hambrusch, S., Walker, E., & Zweben, S. (2017). Generation CS: the growth of computer science. En *ACM Inroads* (pp. 44-50, Vol. 8). Association for Computing Machinery. https://doi.org/10.1145/3084362

Effenberger, T., & Pelánek, R. (2022). Code Quality Defects across Introductory Programming Topics. Association for Computing Machinery. https://doi.org/10.1145/3478431.3499415

Keuning, H., Heeren, B., & Jeuring, J. (2017). Code Quality Issues in Student Programs. Association for Computing Machinery. https://doi.org/10.1145/3059009.3059061

Keuning, H., Jeuring, J., & Heeren, B. (2018). A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. Association for Computing Machinery. https://doi.org/10.1145/3231711

Messer, M., Brown, N. C. C., Kölling, M., & Shi, M. (2024). Automated Grading and Feedback Tools for Programming Education: A Systematic Review. En *ACM Trans. Comput. Educ.* (Vol. 24). Association for Computing Machinery. https://doi.org/10.1145/3636515

Řechtáčková, A., Pelánek, R., & Effenberger, T. (2025). Code Quality Defects in Introductory Programming. En *ACM Trans. Comput. Educ.* (Vol. 25). Association for Computing Machinery. https://doi.org/10.1145/3749996

Stegeman, M., Barendsen, E., & Smetsers, S. (2016). Designing a rubric for feedback on code quality in programming courses. En *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (pp. 160-164). Association for Computing Machinery. https://doi.org/10.1145/2999541.2999555