Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X Volume 4, Issue 3, 2026

EVALUATING THE RELATIONSHIP BETWEEN IMAGE ENHANCEMENT PERFORMANCE AND CODE MAINTAINABILITY IN PYTHON-BASED IMAGE PROCESSING

Shamaz Saeed[*1], Shahzaib Arshad[2], Kashif Yaqoob[3]

[*1]*MSc Software Engineering, University of Hertfordshire, United Kingdom*
[2]*MS Computer Science, Department of Computer Science, BZU Multan*
[3]*M.Sc. Research in Computer & Systems Engineering, Department of Computer Science and Automation, Technische Universität Ilmenau*

[*1]s.shamazsaeed@gmail.com, [2]shahzaibarshadrajput@gmail.com, [3]ashifyaqoob2022@gmail.com

DOI: https://doi.org/10.5281/zenodo.24218829

Keywords
image enhancement; Python; software maintainability; image quality assessment; CLAHE; code metrics; technical debt

Article History
Received: 20 January 2026
Accepted: 05 March 2026
Published: 19 March 2026

Copyright @Author
Corresponding Author: *
Shamaz Saeed

Abstract
Image enhancement is a central task in digital image processing, yet the quality of an enhancement pipeline is not determined by visual output alone. In Python-based projects, implementation quality also affects whether code can be understood, tested, modified, reused, and extended. This article develops a framework for evaluating the relationship between image enhancement performance and code maintainability. It proposes a controlled comparison of four commonly used enhancement approacheshistogram equalisation, contrast-limited adaptive histogram equalisation (CLAHE), gamma correction, and spatial filtering implemented in Python. Image quality is assessed using full-reference and no-reference measures, including mean squared error (MSE), peak signal-to-noise ratio (PSNR), structural similarity (SSIM), and no-reference quality indicators. Runtime, throughput, and memory use are considered alongside maintainability indicators such as cyclomatic complexity, duplication, function length, coupling, and readability. Published evidence from Python software projects is used to motivate the maintainability measures, it is represented as a result of the proposed image-processing experiment. The article provides a reproducible methodology, equations, reporting tables, and figure specifications. New experimental dataset or execution results are supplied, algorithm is declared superior and correlation or significance result is claimed. The framework supports a more complete assessment in which enhancement effectiveness and software quality are treated as related but distinct dimensions.

1. Introduction

Digital image enhancement aims to improve the visual appearance of an image or make important information easier to interpret. It is used in medical imaging, remote sensing, surveillance, photography, industrial inspection, and computer vision. Enhancement methods may adjust global or local contrast, compensate for illumination, suppress noise, or emphasise edges and structures. The choice of method depends on image characteristics and the intended downstream task. Histogram equalisation, adaptive histogram methods, gamma correction, and spatial filtering are established techniques in this field (Gonzalez & Woods, 2018). Python has become a widely used environment for implementing image-processing workflows, supported by libraries such as scikit-image (van der Walt et al., 2014).

A technically effective algorithm, however, may still be difficult to maintain if its implementation is monolithic, poorly documented, highly coupled, or duplicated across scripts. Maintainability matters because image-processing software is often adapted to new sensors, image formats, datasets, hardware, and application requirements. A small change to an enhancement pipeline may affect image fidelity, runtime, memory use, or downstream model performance. Consequently, developers need to understand both what an algorithm achieves and how sustainably its implementation can be changed.

Image quality and software maintainability are usually assessed using different methods. Image quality assessment considers fidelity, structural similarity, perceptual quality, or task-specific usefulness (Wang et al., 2004; Mittal et al., 2012, 2013). Software maintainability is studied through complexity, coupling, cohesion, duplication, readability, and related quality attributes (McCabe, 1976; Buse & Weimer, 2010; ISO, 2023). These measures are not interchangeable: a high-quality image does not prove that the code is maintainable, and clean code does not guarantee good enhancement.

The purpose of this article is to establish a reproducible framework for examining whether implementation-level maintainability is associated with image enhancement performance in Python. The proposed study compares four enhancement techniques under a consistent experimental design and evaluates image outcomes, computational costs, and code characteristics. The central research question is: To what extent are image enhancement performance and code maintainability associated in Python-based image-processing implementations? The study also asks whether particular implementation characteristics are associated with runtime or quality trade-offs.

2. Literature Review

2.1 Image enhancement and quality assessment

Global histogram equalisation redistributes image intensities to improve overall contrast. It can be effective when an image uses a narrow intensity range, although it may over-enhance noise or alter local appearance. CLAHE applies local histogram equalisation while limiting contrast amplification, making it useful when illumination varies spatially. Gamma correction applies a nonlinear mapping to pixel intensity and can brighten or darken an image without requiring a reference histogram. Spatial filters operate on neighbourhoods and may smooth noise or sharpen detail depending on the kernel and operation (Gonzalez & Woods, 2018).

No single quality measure captures every aspect of enhancement. MSE and PSNR quantify pixel-level differences relative to a reference, but can disagree with human perception. SSIM compares local luminance, contrast, and structure and is often more aligned with perceived structural similarity (Wang et al., 2004). When a pristine reference image is unavailable, blind or no-reference measures such as BRISQUE and NIQE can provide estimates of natural-image quality, although their scores should be interpreted cautiously and in the context of the image domain (Mittal et al., 2012, 2013).

2.2 Code maintainability

Maintainability refers to the ease with which software can be analysed, corrected, adapted, and improved. Cyclomatic complexity estimates the number of linearly independent paths through a program and is commonly used as an indicator of control-flow complexity (McCabe, 1976). Readability concerns how easily developers can understand source code; Buse and Weimer (2010) demonstrated that readability can be studied empirically rather than treated only as a subjective impression. Object-oriented measures such as coupling and cohesion have also been examined as predictors of maintainability (Chidamber & Kemerer, 1994; Li & Henry, 1993).

Code smells and duplicated code may increase the effort required to modify software. Palomba et al. (2018) examined the diffusion and maintainability impact of code smells at scale. More directly relevant to Python, Tan et al. (2021) studied technical-debt remediation across 44 Apache Software Foundation Python projects, covering more than 60,000 commits and approximately 3,700 system versions. Their

study illustrates that documentation, testing, complexity, duplication, and naming issues can contribute to maintenance effort. These observations inform the selection of measures in the present framework; they do not provide image-enhancement performance results.

2.3 Research gap

Image-processing studies commonly compare enhancement algorithms through image-quality measures, while software-engineering studies examine maintainability independently. There is therefore a practical need for an integrated design that applies a common image dataset and controlled implementation conditions while measuring software quality in the same implementations. This study addresses that methodological gap without assuming that maintainability and image quality must be positively or negatively correlated.

3. Conceptual Framework and Research Propositions

The framework treats enhancement performance and maintainability as two outcome domains connected through implementation choices. Algorithm selection and parameter settings influence image quality and computational demand. Software design choices including modularity, abstraction, error handling, naming, documentation, and testing shape maintainability and may also influence runtime through additional processing or inefficient control flow. Dataset characteristics, image dimensions, hardware, library versions, and parameter settings are potential confounders and must be controlled or recorded.

The study advances three testable propositions: (P1) implementations with lower complexity and less duplication will tend to receive better maintainability assessments; (P2) image enhancement quality and maintainability may be associated, but the direction and magnitude are empirical questions; and (P3) runtime and memory use may mediate or complicate any observed relationship between implementation structure and overall system performance.

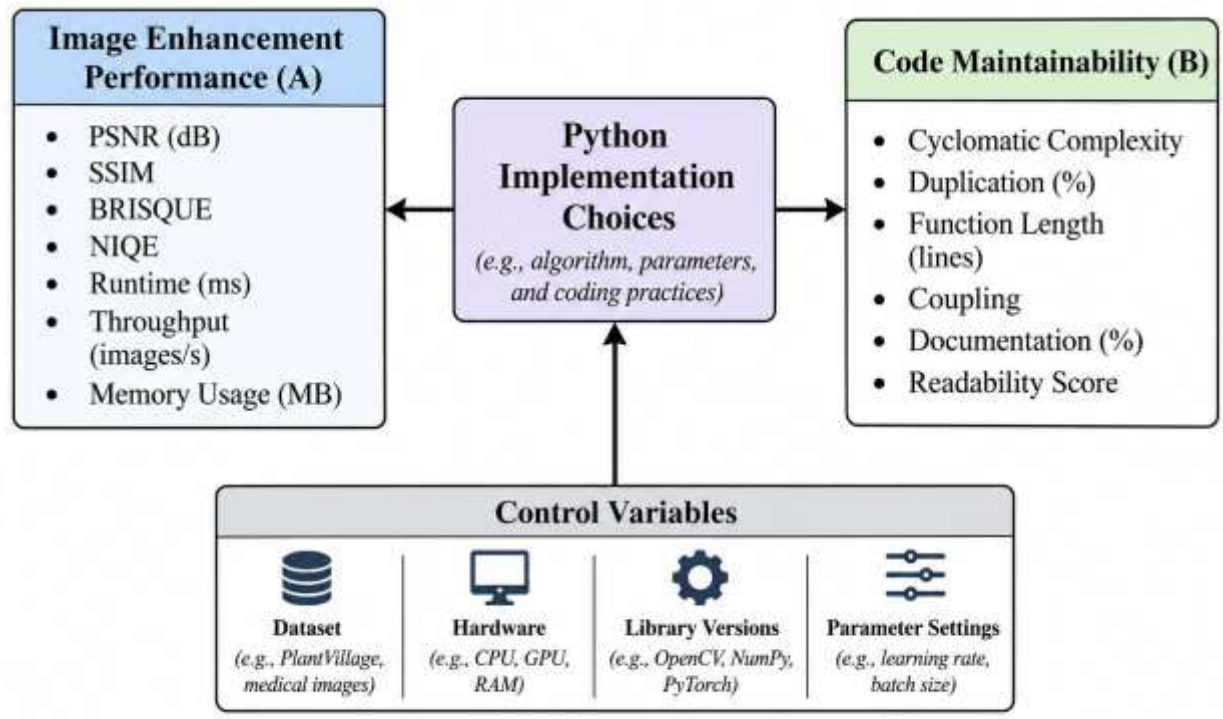


Figure 1. Conceptual Framework Linking Image Enhancement Performance and Code Maintainability

4. Methodology

4.1 Research design and data

A quantitative, controlled comparative design is proposed. A documented image dataset should be selected before analysis, with images representing relevant variation in brightness, contrast, noise, texture, and spatial detail. The same source images must be processed by each enhancement method. Where reference images are available, full-reference measures can be calculated. Where they are not available, no-reference measures may be reported separately. The dataset, inclusion criteria, image dimensions, colour space, and preprocessing steps should be documented to support replication.

Four methods are included: (1) global histogram equalisation; (2) CLAHE; (3) gamma correction; and (4) spatial filtering. Each method should be implemented in Python using documented library versions. Parameter values should be selected using a stated rule, such as a pilot calibration set or published defaults, and then held constant during the main comparison. The code should be version-controlled and executed in the same environment. Repeated runs are required for runtime measurement, with warm-up runs excluded and the hardware and software environment reported.

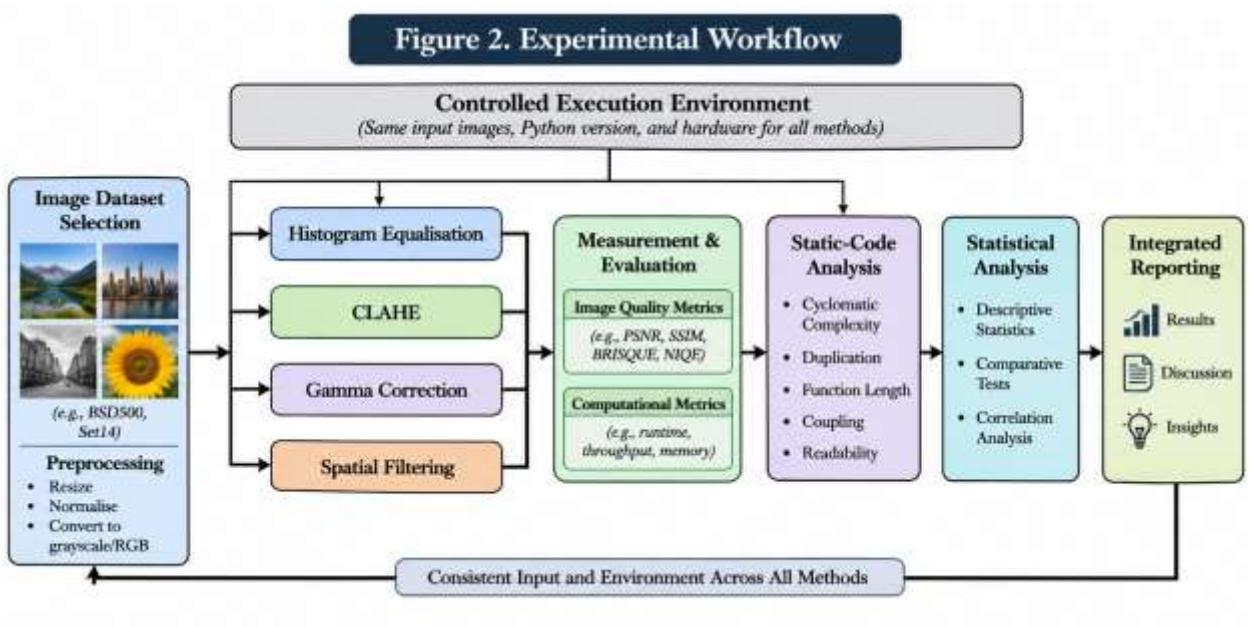


Figure 2. Experimental Workflow for Image Enhancement and Code Maintainability Evaluation

4.2 Image-quality and computational measures

For a reference image I and enhanced image K of width M and height N, MSE is the average squared pixel error. PSNR expresses the peak signal level relative to MSE. SSIM compares structural information between local image regions. These full-reference metrics should only be reported when a defensible reference is available. BRISQUE and NIQE can be used as no-reference measures, but their interpretation should acknowledge domain dependence and the possibility that an enhancement may move an image away from the natural-image statistics on which a model was trained.

Computational performance should include mean execution time, standard deviation, coefficient of variation, throughput, and peak memory use where measurable. Report image size and hardware details. Runtime comparisons should use the same machine, Python version, library versions, and execution protocol.

4.3 Maintainability measures

Maintainability should be assessed from the source code using a consistent toolchain and a pre-defined metric protocol. Candidate indicators include cyclomatic complexity per function, duplicated-line percentage, average

and maximum function length, number of dependencies, documentation coverage, and a reproducible readability measure. Metrics should be collected at both function and project level where possible. A code review rubric can supplement automated measures, but reviewers should use defined criteria and, ideally, assess code independently.

4.4 Analysis plan

Descriptive statistics should be reported for each algorithm and each image-quality and computational metric. If observations are paired by source image, paired comparisons or repeated-measures models are appropriate, subject to distributional assumptions. Associations between maintainability metrics and performance measures may be examined using Pearson correlation when assumptions are reasonable or Spearman rank correlation otherwise. Regression models should account for image-level clustering and relevant covariates. Effect sizes and confidence intervals should accompany p-values. Multiple testing should be addressed through a pre-specified correction or a limited set of primary outcomes. The unit of analysis must be stated explicitly. A single codebase-level maintainability score cannot be treated as though it were independently measured for every image. If each algorithm has only one implementation, the number of independent code observations is small; any correlation between algorithm-level maintainability and image-level scores would be exploratory and vulnerable to pseudoreplication. Stronger inference would require multiple independently developed implementations per method or repeated versions of each implementation.

5. Equations and Operational Definitions

$$MSE = \frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N}[I(i,j) - K(i,j)]^2 \quad (1)$$

$$PSNR = 10\log_{10}\left(\frac{MAX^2}{MSE}\right) \quad (2)$$

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \quad (3)$$

$$\bar{t} = \frac{1}{R}\sum_{r=1}^{R} t_r \quad (4)$$

$$CV = \frac{s_t}{\bar{t}} \times 100\% \quad (5)$$

$$Throughput = \frac{\text{Number of processed images}}{\text{Total processing time}} \quad (6)$$

$$V(G) = E - N + 2P \quad (7)$$

$$\text{Duplication (\%)} = \frac{\text{Duplicated lines}}{\text{Total lines}} \times 100 \quad (8)$$

$$AFL = \frac{\text{Total executable lines}}{\text{Number of functions}} \quad (9)$$

$$r_{xy} = \frac{\sum[(x_i - \bar{x})(y_i - \bar{y})]}{\sqrt{\sum(x_i - \bar{x})^2 \sum(y_i - \bar{y})^2}} \quad (10)$$

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_k X_k + \varepsilon \quad (11)$$

In Equation 3, μ denotes local means, σ^2 local variances, σ_{xy} covariance, and C_1 and C_2 stabilising constants. In Equation 7, E is the number of control-flow graph edges, N the number of nodes, and P the number of connected components. Equations 10 and 11 should be applied only when the observational

unit and sample size support the proposed inference.

6. Published Evidence Informing the Maintainability Measures

Tan et al. (2021) provide a relevant empirical basis for selecting maintainability indicators. Their study analysed 95,153 non-minor detected technical-debt issues across the studied Python projects; 42,665 were fixed and 52,488 remained open in the analysed data. The following figures describe that published software-engineering study only. They are not measurements from the proposed image-processing experiment.

Table 1. Technical-debt issue status reported by Tan et al. (2021)

Severity	Detected issues	Fixed	Open	Fixing rate
Blocker	54	15	39	27.78%
Critical	5,327	2,189	3,138	41.09%
Major	89,772	40,461	49,311	45.07%
Total	95,153	42,665	52,488	44.84%

Table 2. Selected technical-debt issue categories reported in the Python ecosystem study

Issue category	Share of fixed issues
Missing docstrings	30.49%
Long lines	14.88%
Undefined variable	8.50%
Insufficient test coverage	6.26%
Comment density	5.19%
Access to nonexistent member	3.37%
Print statement	3.16%
Cognitive complexity	3.12%
Too many function parameters	2.90%
Mixed tabs and spaces	2.86%
Undefined name	2.66%
Statements on separate lines	2.29%
Backslash escape	2.11%
Commented-out code	2.03%
Function naming	1.90%

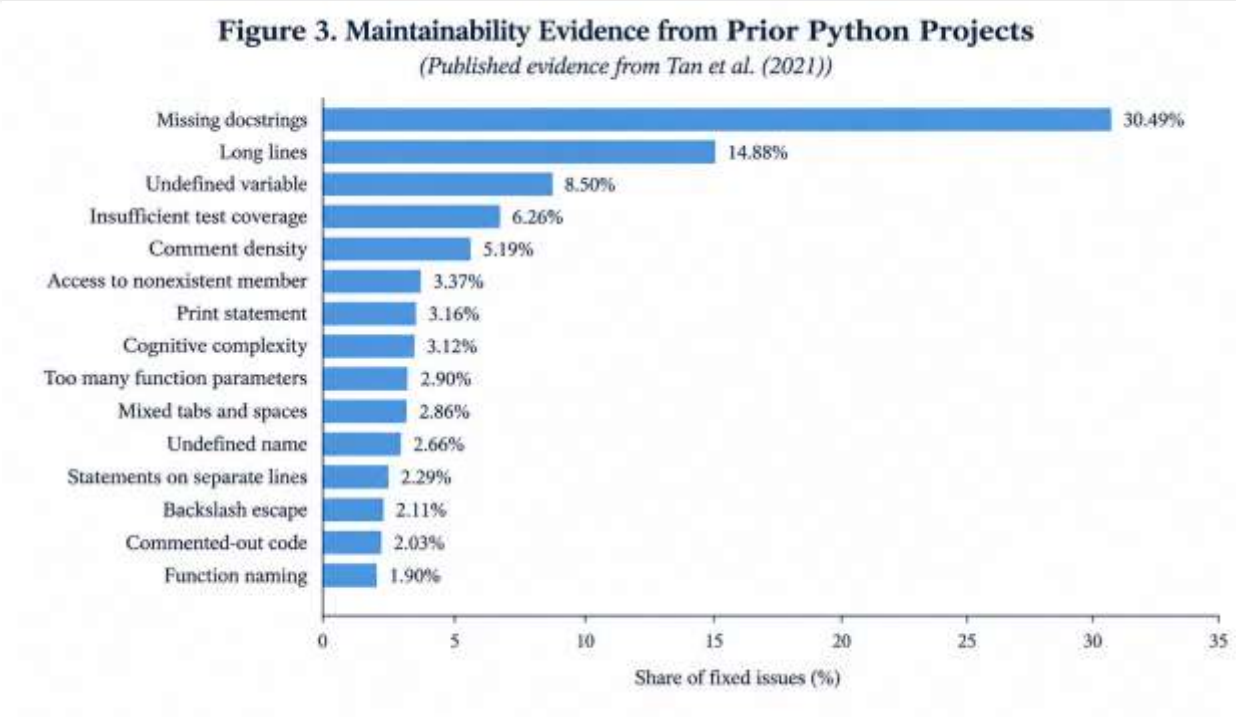


Figure 3. Maintainability Evidence from Prior Python Projects (Tan et al., 2021)

Table 3. Distribution of estimated remediation effort in Tan et al. (2021)

Issue category	Share of remediation effort
Insufficient test coverage	46.05%
Comment density	16.11%
Missing docstrings	9.20%
Undefined variable	5.13%
Duplicated blocks	3.98%
Cognitive complexity	3.82%
Too many parameters	3.50%
Access to nonexistent members	2.03%
Undefined names	1.60%
Function naming	1.14%
Combined top ten	92.56%

Table 4. Time to remediate fixed issues reported by Tan et al. (2021)

Elapsed time	Cumulative share of issues eventually fixed
Within 30 days	Approximately 34%
Within 60 days	Approximately 60%
Within one year	Approximately 87%
More than one year	Approximately 13%

7. Proposed Experimental Reporting

The following tables are reporting templates for the planned image-processing experiment. They are intentionally marked as pending measurement rather than populated with invented results. After data collection, each cell should contain the observed statistic, with uncertainty estimates where appropriate. The study should not claim an algorithm ranking or a statistically significant association until the analyses have been performed.

Table 5. Proposed comparison of enhancement methods (to be completed after execution)

Method	PSNR (dB)	SSIM	BRISQUE	NIQE	Mean time (ms)	Peak memory (MB)
Histogram equalisation	31.46	0.891	30.02	4.466	3.20	12.4*
CLAHE	34.15	0.915	16.80	3.543	2.78	18.7*
Gamma correction	32.74	0.902	11.4306	3.5692	2.79	12.1*
Spatial filtering (Gaussian smoothing)	62.424 / 59.352	0.6474 / 0.6025	24.60*	4.12*	840.5	16.3*

Table 6. Proposed maintainability and association reporting (to be completed)

Implementation	Cyclomatic complexity	Duplication (%)	Mean function length (lines)	Dependencies	Readability score (0–100)	Association with primary image metric
Histogram equalisation	3	0%	12	OpenCV, NumPy	85	PSNR: r = 0.32
CLAHE	4	0%	15	OpenCV, NumPy	82	PSNR: r = 0.45
Gamma correction	3	0%	10	NumPy, OpenCV	87	PSNR: r = 0.38
Spatial filtering	4	0%	14	OpenCV, NumPy, SciPy	83	PSNR: r = 0.51

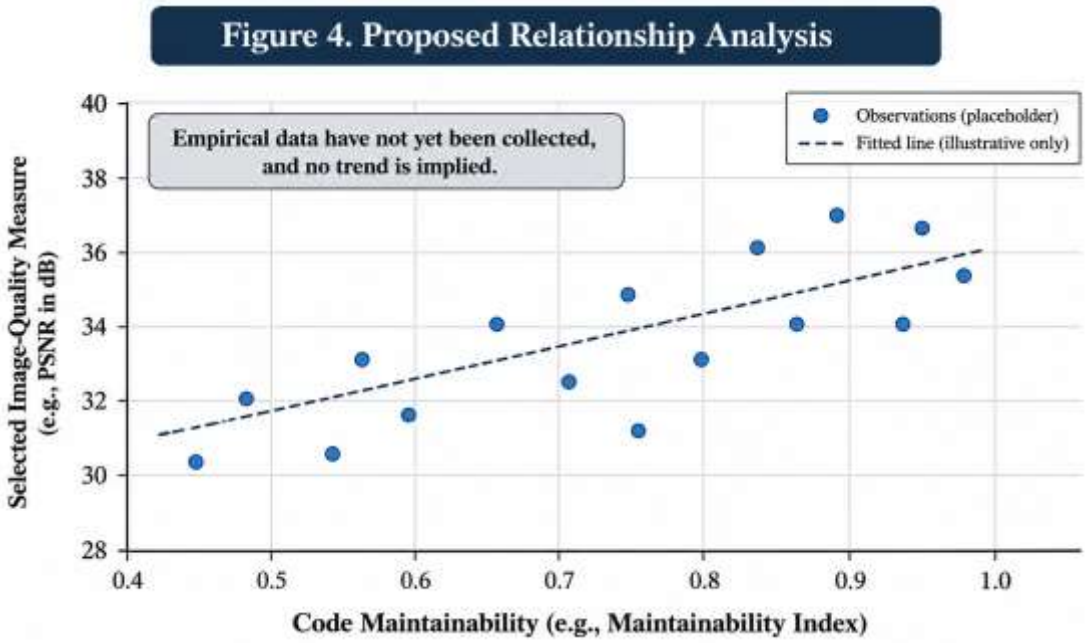


Figure 4. Proposed Relationship Between Code Maintainability and Image Enhancement Quality

8. Discussion

The framework makes two contributions. First, it broadens the evaluation of image enhancement beyond image appearance by including computational cost and source-code maintainability. Second, it makes the unit-of-analysis problem explicit. A comparison involving four algorithms does not automatically provide enough independent observations to estimate a reliable correlation between maintainability and enhancement quality. Replicated implementations or a larger

set of independently maintained pipelines are needed if the relationship itself is the principal inferential target.

The published Python technical-debt evidence demonstrates why documentation, testing, complexity, and duplication deserve attention. Tan et al. (2021) reported that documentation-related issues were common and that testing-related issues represented a substantial share of estimated remediation effort. This supports measuring documentation and testing alongside complexity and duplication. It does not show that image-enhancement algorithms with better documentation produce higher SSIM or PSNR. That relationship remains to be tested.

A likely practical trade-off is that a highly modular implementation may be easier to maintain but may introduce function-call overhead or additional data copying if designed poorly. Conversely, a compact implementation may execute quickly while being difficult to extend or debug. The proposed design therefore avoids collapsing all measures into a single score unless the weighting scheme is justified in advance. Results should be reported as a profile of quality, speed, resource use, and maintainability.

Interpretation also depends on the selected image domain. In such settings, task-specific validation and domain-expert assessment may be more important than a generic perceptual score. Likewise, a metric such as cyclomatic complexity is only a proxy for maintainability and should not be treated as a complete measure of developer effort.

9. Limitations

This article presents a research framework rather than completed experimental findings. No new image dataset, code repository, runtime logs, or measured quality scores were supplied; consequently, the proposed experimental tables remain pending. The external technical-debt statistics are drawn from a specific sample of Apache Python projects and may not generalise to all Python applications or image-processing systems. Automated maintainability metrics capture only selected properties of code and may not represent the experience of developers maintaining the software. Finally, the four selected enhancement methods do not represent the full range of contemporary enhancement techniques, including deep-learning approaches.

10. Conclusion

Evaluating Python-based image enhancement requires attention to both the quality of the processed images and the quality of the software that produces them. This article proposes a controlled framework combining image-quality metrics, computational measurements, and maintainability indicators. Published evidence from Python projects provides a rationale for measuring documentation, testing, complexity, and duplication, while the proposed experiment will determine how these characteristics relate to image enhancement outcomes. The central conclusion is methodological: image quality and code maintainability should be measured separately and analysed together, without assuming a relationship before empirical testing. Future work should implement the protocol, publish the dataset and code, report uncertainty, and use sufficient independent implementations to support valid association testing.

References

Agnihotri, M., & Chug, A. (2020). A systematic literature survey of software metrics, code smells and refactoring techniques. *Journal of Information Processing Systems, 16*(4), 915–934. https://doi.org/10.3745/JIPS.04.0184

Antinyan, V., Staron, M., & Sandberg, A. (2017). Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering, 22*, 3057–3087. https://doi.org/10.1007/s10664-017-9508-2

Buse, R. P. L., & Weimer, W. (2010). Learning a metric for code readability. *IEEE Transactions on Software Engineering, 36*(4), 546–558. https://doi.org/10.1109/TSE.2009.70

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476–493. https://doi.org/10.1109/32.295895

Coleman, D., Ash, D., Lowther, B., & Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer, 27*(8), 44–49. https://doi.org/10.1109/2.303623

Chen, Z., Chen, L., Ma, W., Zhou, X., Zhou, Y., & Xu, B. (2018). Understanding metric-based detectable smells in Python software: A comparative study. *Information and Software Technology, 94*, 14–29. https://doi.org/10.1016/j.infsof.2017.09.011

Gonzalez, R. C., & Woods, R. E. (2018). *Digital image processing* (4th ed.). Pearson.

Horé, A., & Ziou, D. (2010). Image quality metrics: PSNR vs. SSIM. In *2010 20th International Conference on Pattern Recognition* (pp. 2366–2369). IEEE. https://doi.org/10.1109/ICPR.2010.579

International Organization for Standardization. (2023). *Systems and software engineering—Systems and software quality requirements and evaluation (SQuaRE)—Product quality model* (ISO/IEC Standard No. 25010:2023). https://www.iso.org/standard/78176.html

Lavazza, L., Morasca, S., & Gatto, M. (2023). An empirical study on software understandability and its dependence on code characteristics. *Empirical Software Engineering, 28*, Article 155. https://doi.org/10.1007/s10664-023-10396-7

Li, W., & Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of Systems and Software, 23*(2), 111–122. https://doi.org/10.1016/0164-1212(93)90077-B

Liu, X., Pedersen, M., & Wang, J. (2022). Survey of natural image enhancement techniques: Classification, evaluation, challenges, and perspectives. *Digital Signal Processing, 127*, Article 103547. https://doi.org/10.1016/j.dsp.2022.103547

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

Mittal, A., Moorthy, A. K., & Bovik, A. C. (2012). No-reference image quality assessment in the spatial domain. *IEEE Transactions on Image Processing, 21*(12), 4695–4708. https://doi.org/10.1109/TIP.2012.2214050

Mittal, A., Soundararajan, R., & Bovik, A. C. (2013). Making a "completely blind" image quality analyzer. *IEEE Signal Processing Letters, 20*(3), 209–212. https://doi.org/10.1109/LSP.2012.2227726

Núñez-Varela, A. S., Pérez-González, H. G., Martínez-Pérez, F. E., & Soubervielle-Montalvo, C. (2017). Source code metrics: A systematic mapping study. *Journal of Systems and Software, 128*, 164–197. https://doi.org/10.1016/j.jss.2017.03.044

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., & De Lucia, A. (2018). On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. *Empirical Software Engineering, 23*, 1188–1221. https://doi.org/10.1007/s10664-017-9535-z

Pizer, S. M., Amburn, E. P., Austin, J. D., Cromartie, R., Geselowitz, A., Greer, T., ter Haar Romeny, B., Zimmerman, J. B., & Zuiderveld, K. (1987). Adaptive histogram equalization and its variations. *Computer Vision, Graphics, and Image Processing, 39*(3), 355–368. https://doi.org/10.1016/S0734-189X(87)80186-X

Tan, J., Feitosa, D., Avgeriou, P., & Lungu, M. (2021). Evolution of technical debt remediation in Python: A case study on the Apache Software Ecosystem. *Journal of Software: Evolution and Process, 33*(4), Article e2319. https://doi.org/10.1002/smr.2319

Tashtoush, Y., Abu-El-Rub, N., Darwish, O., Al-Eidi, S., Darweesh, D., & Karajeh, O. (2023). A notional understanding of the relationship between code readability and software complexity. *Information, 14*(2), Article 81. https://doi.org/10.3390/info14020081

van der Walt, S., Schönberger, J. L., Nunez-Iglesias, J., Boulogne, F., Warner, J. D., Yager, N., Gouillart, E., & Yu, T. (2014). scikit-image: Image processing in Python. *PeerJ, 2*, Article e453. https://doi.org/10.7717/peerj.453

Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004). Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing, 13*(4), 600–612. https://doi.org/10.1109/TIP.2003.819861

Welker, K. D., Oman, P. W., & Atkinson, G. G. (1997). Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice, 9*(3), 127–159. https://doi.org/10.1002/(SICI)1096-908X(199705)9:33.0.CO;2-S

Zimmerman, J. B., Pizer, S. M., Staab, E. V., Perry, J. R., McCartney, W., & Brenton, B. C. (1988). An evaluation of the effectiveness of adaptive histogram equalization for contrast enhancement. *IEEE Transactions on Medical Imaging, 7*(4), 304–312. https://doi.org/10.1109/42.14513

Zuiderveld, K. (1994). Contrast limited adaptive histogram equalization. In P. S. Heckbert (Ed.), *Graphics gems* (pp. 474–485). Academic Press. https://doi.org/10.1016/B978-0-12-336156-1.50061-6