A Trade-off-Aware Approach for Prioritized Refactoring of Dockerfiles

Délcio Cosme Monjane
Department of Informatics Engineering
School of Engineering of the Technical University of Porto (ISEP)
Porto, Portugal
1211739@isep.ipp.pt

Isabel Azevedo
Department of Informatics Engineering
School of Engineering of the Technical University of Porto (ISEP)
Porto, Portugal
ifp@isep.ipp.pt

Abstract*—Dockerfile smells—violations of recommended practices—are pervasive in open-source projects and degrade image security, build performance, reproducibility and maintainability. Linters such as Hadolint report violations in a flat list, and repair tools such as Parfum apply fixes one at a time; neither indicates which smell deserves attention first, nor exposes the fact that fixing one smell may harm a different quality attribute. This paper presents Docker Prioritizer, a trade-off-aware command-line tool that ranks Dockerfile repairs by their expected impact on four quality attributes, using a Weighted Sum Model over a literature-derived impact matrix combined with developer-defined weights, so that every recommendation makes its trade-offs explicit rather than hiding them inside a single score. The approach was evaluated on 57 Dockerfiles from open-source repositories through an automated build–repair–rebuild pipeline. Across the 24 Dockerfiles where a smell was actually repaired, build time fell by an average of 115 s and image size by 123 MB. On a held-out sample of 11 Dockerfiles not used for calibration, predicted and observed impacts matched in direction for Security (11/11), Performance (10/11) and Maintainability (9/11); reproducibility could not be assessed, as the chosen operationalization (image layer count) did not vary across the corpus.

***Keywords*—Dockerfile smells, automated refactoring, trade-off analysis, prioritization, multi-criteria decision making, container quality**

I. Introduction

Docker has become the de facto standard for packaging and deploying software, and Dockerfiles serve as the executable specification of how a container image is assembled. Empirical studies report that the majority of Dockerfiles in open-source projects contain at least one smell—a violation of a recommended practice that does not break the build but degrades security, performance, maintainability, or reproducibility [1], [6]. Static analysis tools such as Hadolint [13] report these violations as a flat, unranked list, and repair tools such as Parfum [4] fix detected smells one at a time. Neither kind of tool indicates which smell deserves attention first, nor exposes the fact that a given repair may improve one quality attribute while harming another: pinning a package version, for instance, improves reproducibility but removes automatic security patches and adds maintenance overhead.

This paper addresses that gap. It asks which Dockerfile smells have the most significant impact on key quality attributes (RQ1); how a prioritization mechanism can be designed that balances the resulting trade-offs while incorporating developer preferences (RQ2); and what evidence exists for the effectiveness and practical applicability of such an approach (RQ3). We present Docker Prioritizer, a command-line tool that ranks Dockerfile repairs using a Weighted Sum Model over a literature-derived impact matrix, so that every recommendation makes its trade-offs explicit rather than hiding them inside a single score.

This paper's contributions are:

- A literature-derived impact matrix scoring ten repair actions against four quality attributes—Security, Performance, Maintainability, and Reproducibility.
- A Weighted Sum Model prioritisation mechanism that keeps every attribute's contribution to the final score individually inspectable.
- Docker Prioritizer, an open-source implementation (available at https://github.com/delcioMonjane/dmei2) evaluated end-to-end on 57 real-world Dockerfiles.

II. Background and Related Work

A container packages an application with its dependencies into a lightweight, portable unit, and a Dockerfile defines how that unit is built, layer by layer; layer-level caching makes instruction order and content directly responsible for build performance, image size, and reproducibility, while instructions such as USER and ADD directly affect the resulting security posture [1], [3]. Tool support for Dockerfile quality falls into two categories. Detection-only linters—Hadolint [13] and the shell-focused Binnacle [8]—flag violations but treat every finding as equally urgent. Repair-capable tools go further: Parfum [4] automatically fixes efficiency-related smells but reports a build-failure rate near 7%, and DockerCleaner [5] targets security-related smells specifically. None of these four tools ranks detected smells by their expected impact, and none exposes the trade-offs that a given repair introduces across other quality attributes—the gap this paper addresses. Prior work has treated prioritization in a narrower sense: Doctor [9] reorders Dockerfile instructions according to their effect on rebuild efficiency, and empirical work on Docker refactoring has framed smells as a form of technical debt [10]; neither ranks individual repairs across competing quality attributes. Table I summarises these capabilities.

TABLE I. CAPABILITIES OF DOCKERFILE QUALITY TOOLS

Tool	Detect	Repair	Rank by impact	Expose trade-offs
Hadolint [13]	yes	no	no	no
Binnacle [8]	yes	no	no	no
Parfum [4]	yes	yes	no	no
DockerCleaner [5]	yes	yes (security)	no	no
This work	via [4]	via [4]	yes	yes

III. Trade-off-Aware Prioritization

A. Quality Impact Matrix

The artifact underlying Docker Prioritizer is an impact matrix scoring repair actions against four quality attributes—Security, Performance, Maintainability, and Reproducibility—on a −5 to +5 scale. Values were derived from a systematic review of the Dockerfile-smell literature [1]–[3], [6], synthesizing each study's reported or implied effect of a given repair action on a given attribute. Table II reports the six actions most frequently triggered in our evaluation corpus (Section IV-B); the complete ten-action matrix is reported in the accompanying dissertation. The matrix was subsequently checked against 28 real-world Dockerfiles repaired in an earlier evaluation round; this calibration step did not surface evidence strong enough to justify revising any literature-derived value, so the matrix reported here should be read as literature-derived and empirically checked, rather than empirically calibrated.

TABLE II. QUALITY IMPACT MATRIX (SIX MOST FREQUENT ACTIONS)

Repair action	Sec.	Perf.	Maint.	Repro.
Merge install commands	0	+4	+2	0
Add --no-install-recommends	0	+4	0	0
Remove package caches	0	+5	+1	0
Run as non-root user	+5	0	+1	0
Remove platform constraints	0	0	+3	+2
Pin dependency versions	−2	0	−1	+5

As a worked example, the "Pin dependency versions" row illustrates how the four signals were derived. Reproducibility (+5) follows directly from the build-determinism argument common to empirical studies of this smell [1], [6]: an unpinned dependency can resolve to a different version on every build, so pinning is reported as the primary lever for reproducible images. Security (−2) reflects the converse and equally well-documented concern raised in the same studies: a pinned dependency stops receiving upstream security patches automatically, so the version must be bumped manually to remain current. Maintainability (−1) follows from that same manual-bump burden, scored smaller than the Security cost because it is a recurring but low-effort task rather than a latent vulnerability. Performance was left at 0 because none of the reviewed studies report a build-time or image-size effect attributable to version pinning itself, as distinct from cache- or layer-related smells. Where sources disagreed on direction for a given cell, or reported no effect at all, the value was set to 0 rather than resolved by majority vote, since the matrix is intended to flag confidently-known effects rather than average out disagreement; ties of this kind occurred only for Performance and Reproducibility on security-focused actions, and are visible as 0 entries in Table II.

B. Priority Score

Given a detected smell s and the set of quality attributes Q = {Security, Performance, Maintainability, Reproducibility}, Docker Prioritizer computes a priority score as a Weighted Sum Model:

$$Priority(s) = \Sigma_{q \in Q}\, w_q \cdot I(s, q) \quad (1)$$

where w_q is a developer-supplied weight for attribute q and $I(s, q)$ is the corresponding entry in the impact matrix. We selected WSM [12] over alternatives such as AHP or TOPSIS [11] because it requires neither pairwise comparison matrices nor an additional normalization step, and because it keeps each attribute's contribution to the final score individually inspectable—a property we consider essential for a recommendation whose purpose is to expose trade-offs rather than hide them.

C. Tool Implementation

Docker Prioritizer is implemented in Python and distributed as a Poetry-managed command-line tool. As shown in Fig. 1, a Dockerfile is first passed to Parfum [4], which performs detection and, where possible, automated repair; each detected smell's per-attribute impact is retrieved from the matrix and combined with a YAML file of developer-supplied weights via (1) to produce a single ranked list. Running `docker-prioritizer prioritize` on an example Dockerfile with an equal-weight profile produces output such as:



Fig. 1. Docker Prioritizer pipeline: an external detector feeds a literature-derived impact matrix; developer-defined weights turn it into a ranked list with trade-offs preserved.

```
#1  [score -6.00]  DL3002 - running as root
     Security: -5  Maintainability: -1
...
#4  [score -5.00]  DL3020 - ADD instead of COPY
     Performance: +1  Maintainability: -4
     Reproducibility: -2
     Trade-off: improves Performance but reduces
     Maintainability and Reproducibility.
```

Beyond ranking, every entry with a genuine cross-attribute cost is annotated with the specific trade-off in natural language, rather than left implicit in the numeric scores alone.

IV. Evaluation

A. Design and Setup

Following a Design Science Research approach [7], we evaluated Docker Prioritizer on 57 Dockerfiles collected from open-source GitHub repositories through an automated build–repair–rebuild pipeline: each Dockerfile is built to establish a baseline, passed to Parfum for detection and repair, and rebuilt; each build runs three times from a cold cache, with the median retained to reduce measurement noise. The corpus was collected in May 2026 by scraping public GitHub repositories and comprises 57 Dockerfiles drawn from 55 distinct projects, since some ship separate backend and frontend images; each Dockerfile is treated as an independent unit of analysis. All builds ran on a single machine with Ubuntu 20.04.4 LTS and Docker Engine 20.10.6. The evaluation addresses two sub-questions: EQ1, whether automated refactoring measurably improves the target quality attributes, and EQ2, whether the direction and magnitude of observed changes agree with the impact matrix's predictions. Two further sub-questions on practical acceptance—maintainer responses to Dockerfiles submitted as pull requests—could not be answered within the study's timeline and are left for future work. A complementary analysis, reported in Section IV-D, examines how sensitive the resulting ranking is to the developer's choice of weight profile.

B. Effect of Refactoring on Quality Attributes (EQ1)

Of the 57 Dockerfiles, 40 completed the full build–repair–rebuild cycle: 16 had no detectable smell and required no repair, and 24 had at least one smell successfully repaired

and verified; the remaining 17 did not complete the cycle (Section V). Image size is the attribute for which the impact of Dockerfile smells has been most directly measured in prior work [2]. Table III reports average build-time and image-size change across all 40 completed Dockerfiles and, separately, across the 24 where a repair actually took place. Restricting to the repaired subset raises the average build-time reduction from 66.1 s (−30.9%) to 115.2 s, and the average image-size reduction from 73.8 MB (−16.9%) to 123.0 MB—roughly double—confirming that the smaller all-Dockerfile figure reflects dilution by no-op cases, together with measurement variance in the unrepaired files, which averaged a 7.7 s regression across repeated cold builds, rather than a weak underlying effect. Both effects are substantially larger than those measured in an earlier 28-Dockerfile round (−18.5% and −1.5% respectively), which is consistent with a larger and more varied sample rather than with a change in the approach. Among the abstract quality scores, Performance again showed the largest average gain (+7.58), followed by Security (+0.90) and Maintainability (+0.55), while Reproducibility did not change on average. Layer count did not change in any of the 40 Dockerfiles, despite the image-size reduction. The reduction therefore comes from cleaner content inside the existing layers rather than from fewer layers, which revises the more general explanation offered in the earlier round of this evaluation. This is discussed further in Section V. A second null result concerns security: the pipeline also recorded the number of CRITICAL and HIGH vulnerabilities reported by Trivy [14] for each image, and this count fell only from 2.45 to 2.42 on average (−1.2%) across the 40 completed Dockerfiles, while the aggregate abstract Security score rose from −0.90 to 0.00, meaning that every catalogued security smell had been removed. Repairing the security smells that current detectors recognise therefore did not translate into measurably fewer known vulnerabilities in the resulting images. Only one of the 40 images showed any reduction at all, and by a single vulnerability. The likely reason is that smell repair as implemented here addresses configuration and packaging hygiene, whereas the findings Trivy reports are dominated by vulnerabilities inherited from the base image, which no Dockerfile-level rewrite of this kind removes. Across the 57 Dockerfiles, 88 individual smell instances spanning 9 smell types were logged in the 36 files with at least one detected smell; the four most frequent types—each with a Performance component—accounted for 74% of all instances. Table IV gives the full distribution.

TABLE III. EFFECT OF REFACTORING ON BUILD TIME AND IMAGE SIZE (EQ1)

Metric	All completed (n=40)	Repaired subset (n=24)
Build time Δ (s)	−66.1 (−30.9%)	−115.2
Image size Δ (MB)	−73.8 (−16.9%)	−123.0

TABLE IV. SMELL TYPE DISTRIBUTION ACROSS THE CORPUS (88 INSTANCES, 36 FILES)

Smell type	Instances	Primary attribute
pipUseNoCacheDir	22	Performance
ruleMoreThanOneInstall	17	Performance / Maintainability
aptGetInstallUseNoRec	13	Performance
DL3046	13	Performance / Maintainability
npmCacheCleanAfterInstall	8	Performance
tarSomethingRmTheSomething	7	Performance
DL3029	5	Reproducibility / Maintainability
gpgUseBatchFlag	2	Maintainability
DL3002	1	Security

C. Agreement Between Predicted and Observed Impacts (EQ2)

For each Dockerfile with at least one logged smell, the predicted per-attribute impact of its smells was summed and compared against the observed change in that attribute. Of the 24 repaired Dockerfiles with a logged smell record, 13 belonged to the 28 repositories used in the earlier verification round described in Section III-A; although that round did not alter any matrix value, they were excluded to keep the validation strictly held-out, leaving 11 Dockerfiles for held-out validation. Table V reports the agreement between predicted and observed impact on this held-out sample. Because the observed per-attribute change is itself obtained by summing matrix values over the smells still present after repair, EQ2 measures how completely the predicted repairs were realised rather than whether the matrix values are themselves correct; Section V returns to this limitation.

TABLE V. PREDICTED VS. OBSERVED IMPACT, HELD-OUT SAMPLE (EQ2, N=11)

Attribute	Direction match	Exact magnitude	r
Security	11/11 (100%)	11/11 (100%)	1.00*
Performance	10/11 (91%)	9/11 (82%)	0.73
Maintainability	9/11 (82%)	9/11 (82%)	0.63
Reproducibility	9/11 (82%)	9/11 (82%)	undef.

**See discussion below for a caveat on this coefficient.*

Security predictions matched the observed outcome exactly in all 11 cases, both in direction and magnitude. Performance matched direction in 10 of 11 cases and exact magnitude in 9, and Maintainability matched direction in 9 of 11—a reasonably strong relationship given the small sample. Reproducibility is the weakest result: in 2 of the 11 Dockerfiles a nonzero impact was predicted but the observed score did not change, and because observed reproducibility has almost no variance in this sample, its correlation is undefined rather than merely low. Because observed reproducibility was constant at zero across the corpus (Section V), most of Reproducibility's 9 direction matches are trivial zero-predicted, zero-observed pairs rather than confirmed nonzero predictions; this row of Table V should not be read as validating the matrix's reproducibility values.

We report match counts as the primary evidence in Table V; correlation coefficients are included for reference, but the Security coefficient in particular should be interpreted with care, since a high proportion of true-negative (no security-relevant smell present) cases can inflate an aggregate correlation without reflecting genuine predictive discrimination on nontrivial cases. Of the 11 held-out Dockerfiles, only 5 had a nonzero security effect, predicted and observed alike, so r = 1.00 is driven in part by correctly predicted null cases rather than solely by discrimination across nontrivial impacts. A plausible explanation for the residual mismatches on other attributes is that a smell can be detected and logged with its predicted impact without Parfum's repair step being able to apply a fix for it.

D. Sensitivity to Developer Weight Profiles

The prioritization mechanism is only useful if developer weights actually change what it recommends. Because the priority score is a weighted sum over the impact matrix, this can be examined retrospectively on the data already collected, without a further build cycle. We re-ranked the 88 logged smell instances under two profiles: the equal-weight profile used throughout the evaluation, and a maintainability-first profile — representing a developer who values long-term upkeep over short-term build-time gains — that assigns Maintainability a weight of 2.0 and Performance a weight of

0.5. The top-ranked smell changed in 6 of the 24 Dockerfiles with more than one detected smell (6 of 36 overall), and the set of smells in the top three changed in 1 of the 6 Dockerfiles with more than three; in the remaining 30 files, the top three is the complete set of detected smells and therefore cannot change under any weight profile. Table VI reports the mean rank position, across all files containing each type, for the five most frequent smell types under the two profiles: types with a genuine Maintainability component shift (ruleMoreThanOneInstall, DL3046) hold or gain priority, while types that are purely Performance-driven (aptGetInstallUseNoRec) lose it, as their dominant attribute's weight is halved. Across the full nine-type set, the pattern is sharper still: npmCacheCleanAfterInstall falls the furthest (mean rank 1.12 to 2.25, its Performance-heavy profile losing the most weight), and DL3029 rises the most (2.00 to 1.40, driven by its Maintainability component).

TABLE VI. RANK OF THE FIVE MOST FREQUENT SMELL TYPES UNDER TWO WEIGHT PROFILES

Smell type	Rank (equal)	Rank (maint.-first)	Shift
pipUseNoCacheDir	1.50	1.50	0.00
ruleMoreThanOneInstall	2.41	2.06	-0.35
aptGetInstallUseNoRec	3.15	3.31	+0.16
DL3046	2.31	2.23	-0.08
npmCacheCleanAfterInstall	1.12	2.25	+1.13

V. Threats to Validity

Internal validity is limited by measurement scope: reproducibility was operationalized as image layer count, which did not vary in any of the 40 completed Dockerfiles—a null result that leaves the corresponding row of Table V essentially untested by this corpus, rather than confirmed. Construct validity is further limited by the small held-out sample (n = 11) used for EQ2; the agreement observed there is encouraging but not conclusive at this scale. A further construct-validity limitation concerns what EQ2 can establish. Both the predicted and observed per-attribute values are derived from the same impact matrix. Thus, a Dockerfile whose smells are all repaired yields an observed change equal to the prediction by construction, and the mismatches reported above arise where Parfum could not apply a fix. EQ2 therefore validates the completeness of the repair step against the prediction, not the correctness of the matrix values themselves. An independent validation would have to compare predictions against measured outcomes — build time, image size or vulnerability counts — rather than against scores derived from the matrix itself. The vulnerability result reported in Section IV-B is a first step in that direction, and for Security it is not encouraging.

External validity is constrained by the evaluation's dependency on Parfum, as the sole detection and repair oracle: 7 of the 57 Dockerfiles exhibited a build regression after repair, and in an earlier 28-Dockerfile batch, 4 of 5 such regressions shared an identical parser error on the FROM instruction, pointing to a systematic defect in Parfum's own repair logic rather than in the prioritization layer evaluated here. This observed regression rate (7/57, 12.3%) is markedly higher than the near-7% failure rate Parfum's own authors report on their evaluation corpus [4]; the FROM-instruction defect identified above is a plausible explanation for the difference. A further 10 Dockerfiles could not be built even before any repair was attempted, independent of this work. Because repair-induced regressions, rather than an inability to parse unfamiliar Dockerfiles, account for 7 of the 17 non-completions, any pull request generated by this pipeline must be preceded by a successful local rebuild of the repaired file; the pipeline enforces this implicitly, since only Dockerfiles that rebuild successfully reach the submission step. Finally, the corpus of 57 Dockerfiles, though larger than an earlier 28-file round, remains a convenience sample from open-source GitHub repositories and a work in progress; the practical-acceptance questions of whether maintainers would merge these repairs (EQ3, EQ4) could not be assessed within the study's timeline, since responses to the pull requests submitted as part of this evaluation had not been received.

VI. Conclusion and Future Work

In direct answer to the questions posed in Section I: RQ1 is answered by the impact matrix and its empirical confirmation in Section IV-B, where package-manager and cache-related smells are the most frequent and belong to the attribute category, Performance, in which the largest aggregate effects were measured. RQ2 is answered by the Weighted Sum Model mechanism itself, which distinguishes itself from prior tools by keeping every quality attribute's contribution to the ranking individually inspectable rather than collapsed into an opaque score. This is measurable, not just a design claim: re-ranking under a maintainability-first profile changes the top-ranked smell in 6 of the 36 Dockerfiles with a detected smell (Section IV-D). RQ3 is answered with qualified confidence: EQ1 and EQ2 provide positive evidence of effectiveness and prediction validity for Security, Performance and Maintainability, but practical acceptance (EQ3, EQ4) remains unanswered, since maintainer responses to the submitted pull requests had not been received at the time of writing.

This paper presented Docker Prioritizer, a trade-off-aware prioritization layer for Dockerfile refactoring that combines a literature-derived impact matrix with a Weighted Sum Model to rank detected smells by expected, developer-weighted impact while keeping every cross-attribute trade-off explicit. Evaluated on 57 real-world Dockerfiles, the approach reduced average build time by 30.9% and image size by 16.9% across all 40 completed files, rising to 115.2 s and 123.0 MB in absolute terms when restricted to the 24 files where a repair actually occurred; its predictions agreed with observed outcomes exactly for Security and strongly for Performance and Maintainability on a held-out validation sample.

Repaired Dockerfiles have been submitted as pull requests to open-source projects as ongoing work; maintainer responses had not been received at the time of writing, so this is reported as evidence of practical usability rather than as a completed acceptance result. Future work includes collecting these outcomes once available, fixing the FROM-instruction defect identified in Parfum's repair logic and growing the held-out validation sample.

References

[1] Y. Wu, Y. Zhang, T. Wang, and H. Wang, "Characterizing the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study," IEEE Access, vol. 8, pp. 34127–34139, 2020.

[2] T. Durieux, "Empirical Study of the Docker Smells Impact on the Image Size," in Proc. ICSE 2024, pp. 2568–2579.

[3] G. Rosa, S. Scalabrino, G. Robles, and R. Oliveto, "Not all Dockerfile Smells are the Same: An Empirical Evaluation of Hadolint Writing Practices by Experts," in Proc. MSR 2024, pp. 231–241.

[4] T. Durieux, "Parfum: Detection and Automatic Repair of Dockerfile Smells," arXiv:2302.01707, 2023.

[5] Q.-C. Bui, M. Laukötter, and R. Scandariato, "DockerCleaner: Automatic Repair of Security Smells in Dockerfiles," in Proc. ICSME 2023, pp. 160–170.

[6] G. Rosa, F. Zappone, S. Scalabrino, and R. Oliveto, "Fixing Dockerfile smells: an empirical study," Empirical Software Engineering, vol. 29, no. 5, 2024.

[7] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research," Journal of Management Information Systems, vol. 24, no. 3, pp. 45–77, 2007.

[8] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, "Learning from, Understanding, and Supporting DevOps Artifacts for Docker," in Proc. ICSE 2020, pp. 38–49.

[9] Z. Zhu et al., "Doctor: Optimizing Container Rebuild Efficiency by Instruction Re-orchestration," Proc. ACM Softw. Eng., vol. 2, no. ISSTA, pp. 1–23, 2025.

[10] E. Ksontini, M. Kessentini, T. N. do Ferreira, and F. Hassan, "Refactorings and Technical Debt in Docker Projects: An Empirical Study," in Proc. ASE 2021.

[11] D. L. Olson, "Comparison of weights in TOPSIS models," Math. Comput. Model., vol. 40, no. 7–8, pp. 721–727, 2004.

[12] D. Handoko, S. D. Nasution, and H. Nurdiyanto, "Application of Weight Sum Model (WSM) in Determining Special Allocation Funds Recipients," Int. J. Informatics and Computer Science, 2017.

[13] Hadolint: a smarter Dockerfile linter. [Online]. Available: https://github.com/hadolint/hadolint

[14] Trivy: a vulnerability scanner for containers. [Online]. Available: https://github.com/aquasecurity/trivy