Dockerfile Carbon Optimizer

Alexandru Verhovetchi, Ion Tulei, Horia Zaharia, Dragos Erhan, Joost Weerheim
CS4575 TU Delft

ABSTRACT

The growing environmental impact of software systems has motivated efforts to reduce the carbon footprint of application development and deployment. This paper introduces the Dockerfile Carbon Optimizer (DCO), a command-line tool that analyzes Dockerfiles for energy-wasteful patterns and applies automated fixes to produce optimized versions with carbon-savings estimates. DCO implements six detection rules based on common Dockerfile inefficiencies: oversized base images, uncombined `RUN` layers, development dependencies left in production images, missing `.dockerignore` files, absent multi-stage builds, and unpinned base image tags. Carbon impact is estimated using the Aslan et al. network energy model [2] combined with IEA grid intensity factors [8], parameterized by the image pull frequency drawn from Docker Hub. We validate DCO against ten real-world GitHub repositories across five programming languages. The results show image-size reductions of up to 366 MB per image for repositories where the base image swap rule triggers, with measurable reductions in build energy consumption confirmed via CodeCarbon [9] hardware energy monitoring.

1 INTRODUCTION

Container technology such as Docker has become the standard deployment method for modern software, with Docker images pulled billions of times per month across public registries [10]. Every pull transfers image layers over the network, causing energy consumption and emissions proportional to the amount of data transferred. Larger images mean more data per pull, more storage on every host, and longer build times.

Despite this, most Dockerfile tooling focuses on correctness and security rather than sustainability. Tools such as *hadolint* [3] and *dockle* [6] warn about shell-scripting pitfalls and best practices, but do not estimate the carbon cost of identified issues and do not produce fixed Dockerfiles. The existing academic literature quantifies the emissions attributable to internet data transmission [2] and data-center energy use [10], yet stops short of providing actionable, automated tooling that closes the loop from detection to fix inside a developer's workflow.

We address this gap with the Dockerfile Carbon Optimizer (DCO). DCO is an open-source Python CLI tool that (i) parses a Dockerfile into a structured instruction tree, (ii) applies six detection rules that flag energy-wasteful patterns, (iii) estimates the monthly CO_2 cost of each finding using a network-transfer energy model calibrated to Aslan et al.'s published figures [2] and IEA grid intensity factors [8], and (iv) automatically rewrites the Dockerfile to eliminate four of the six detected patterns (with a fifth available via a `-force` flag for cases that require user verification). The tool integrates into any continuous-integration pipeline: a single `dco analyze` invocation produces a machine-readable JSON or CSV report, and `dco fix` produces a corrected Dockerfile with no manual intervention.

We validate DCO against ten real-world GitHub repositories spanning five programming languages (Python, Node.js, Java, Go, PHP). Using intentionally un-optimized Dockerfiles, we measure image size, build time, and energy consumption before and after applying DCO's fixes. The results confirm measurable size reductions (up to 366 MB) and energy savings for repositories where high-impact rules such as base image replacement trigger.

The remainder of this paper is organized as follows. section 2 reviews related work on network energy modeling and Dockerfile analysis tools. section 3 describes DCO's architecture, carbon model, and detection rules. section 4 evaluates the tool against ten real-world repositories and quantifies the savings. section 5 discusses the implications and limitations of the results. section 6 outlines planned extensions. section 7 concludes the paper.

2 BACKGROUND AND RELATED WORK

2.1 Energy and Carbon Footprint of Data Transmission

Quantifying the carbon cost of transmitting data over the internet requires a model that maps gigabytes transferred to kilowatt-hours consumed and then to grams of CO_2 emitted. Aslan et al. [2] provide the most cited model: they estimate the electricity intensity of internet data transmission at approximately 0.06 kWh/GB in 2015, and show that this figure halves roughly every two years as network infrastructure becomes more efficient. Extrapolating to 2026 using an 11-year doubling period yields approximately 0.0013 kWh/GB, a figure we adopt as the default in DCO's carbon model.

Guénnebaud [7] revisit the same question using a bottom-up methodology, distinguishing between intensity indicators (kWh/GB) and absolute estimates. Their work reinforces the trend of decreasing intensity but cautions against using intensity figures in isolation, as absolute energy consumption of the internet has continued to grow due to traffic volume increases. Our tool exposes both the intensity parameter and the pull-frequency parameter so that users can verify DCO's estimates.

2.2 Data Center and Cloud Energy Use

Masanet et al. [10] recalibrate global data-center energy-use estimates and find that despite a factor-of-six increase in compute workloads between 2010 and 2018, data-center electricity consumption grew by only 6%. This efficiency gain is largely attributable to hyperscale cloud providers and server virtualization. Container images hosted in registries such as Docker Hub reside in these data centers; smaller images reduce storage footprint and the energy cost of layer decompression on pull.

2.3 Carbon Intensity of Electricity Grids

Grid carbon intensity (measured in gCO_2/kWh) varies widely by geography and time. We use the IEA Emission Factors dataset [8] combined with the Ember Global Electricity Review 2024, which reports a world-average grid intensity of approximately 445 gCO_2/kWh. DCO accepts a `-region` argument to override this default with a

country-specific factor from the same dataset. The Climatiq platform [5] and `CarbonFootprint.com` [4] provide complementary per-country time series that can be used for more granular analysis.

2.4 Dockerfile Analysis and Linting Tools

Several open-source tools analyze Dockerfiles for issues unrelated to energy: hadolint [3] is a Haskell-based linter that flags shell-scripting anti-patterns (e.g., missing `-no-install-recommends`) and structural issues. dockle [6] focuses on CIS benchmark compliance and image hardening. Neither tool estimates carbon costs or produces fixed Dockerfiles. Trivy [1] scans images for vulnerability and misconfiguration but likewise does not address energy efficiency.

DCO is the first tool, to our knowledge, that combines rule-based Dockerfile analysis with automated carbon-cost estimation and automated remediation in a single developer-facing CLI.

3 METHODOLOGY

3.1 System Overview

DCO is structured as a pipeline of four stages: *parse*, *detect*, *estimate*, and *fix*.

(1) **Parse.** The `dco.parser` module wraps the `dockerfile-parse` library [11] to produce a `ParsedDockerfile` dataclass containing the raw text, a list of instruction dictionaries (each with `instruction`, `value`, `startline`, and `endline` fields), the resolved base image string, and all `FROM` instructions for multi-stage awareness.
(2) **Detect.** Each detection rule is an independent Python class decorated with `@register`, which adds it to a global registry at import time. The CLI calls `discover_rules()` to auto-import every module in the `dco/rules/` package, then iterates the registry and calls `rule.check(parsed, context)` for each rule. Rules return a list of `Finding` objects.
(3) **Estimate.** For each `Finding`, the output layer applies the carbon model (see subsection 3.2) using the `size_saved_mb` field to produce an estimated monthly CO_2 saving.
(4) **Fix.** The `dco.fixer` module collects the `FixAction` objects embedded in auto-fixable `Findings`, sorts them by line number in descending order (bottom-to-top application prevents line-offset corruption), and replaces the targeted line ranges in the original content string. The result is validated by re-parsing with `dockerfile-parse` before being written to disk.

The tool is packaged as a `dco` entry-point script and exposes four CLI commands: `analyze`, `fix`, `batch`, and `info`. The `batch` command recursively locates all Dockerfiles in a directory and runs the full pipeline on each, emitting a summary of total findings and aggregate savings. The `info` command queries the Docker Hub API for a given image and displays its pull count, star count, and description, helping users gauge the real-world pull frequency of their base images.

3.2 Carbon Impact Modeling

DCO estimates the monthly CO_2 saving attributable to a given image-size reduction using the following formula, derived from the Aslan et al. network energy model:

$$\mathrm{CO}_2^{\mathrm{month}} = \Delta S_{\mathrm{GB}} \times F_{\mathrm{pull}} \times E_{\mathrm{net}} \times I_{\mathrm{grid}} \tag{1}$$

where:

- ΔS_{GB} is the estimated image-size reduction in gigabytes introduced by fixing the finding;
- F_{pull} is the number of image pulls per month, defaulting to 100,000 (configurable via `-pulls-per-month`);
- E_{net} is the network electricity intensity in kWh/GB, computed as $0.06 \times 0.5^{(Y-2015)/2}$ following the halving model of Aslan et al. [2], where Y is the current year (0.0013 kWh/GB for 2026);
- I_{grid} is the grid carbon intensity in gCO_2/kWh, defaulting to 445 gCO_2/kWh (IEA/Ember 2024 world average [8]).

The result of Equation 1 is in grams of CO_2 saved per month. All four parameters are exposed as CLI options or constants in `dco/config.py`, making the model fully transparent and reproducible.

CodeCarbon [9] is available as an optional dependency for users who wish to measure the *build-time* energy cost directly using hardware performance counters, complementing the network-transfer estimate above. The `BuildEnergyTracker` class in `dco.carbon.build` wraps a Docker build command (or any callable) with a CodeCarbon `EmissionsTracker`, recording energy in kWh and CO_2 in kg for the duration of the build.

3.3 Optimization Strategies

Table 1 summarizes the six detection rules.

Table 1: DCO detection rules, auto-fix capability, and size-saving approach.

ID	Name	Auto-fix	Size saved
DCO001	Oversized base image	`-force`	Data-driven (image DB)
DCO002	Uncombined RUN layers	Yes	0 MB (layer hygiene)
DCO003	Dev deps in production	Yes	Per-package (package DB)
DCO004	Missing .dockerignore	Yes	Data-driven (dir scan)
DCO005	Missing multi-stage	No	Data-driven (image DB)
DCO006	Unpinned base image tag	Yes	0 MB (reproducibility)

DCO001 - Oversized Base Image. Many Dockerfiles use full-distribution base images when a slim variant is available. For example, `python:3.12` has a compressed transfer size of approximately 392 MB, while `python:3.12-slim` is only about 41 MB, a saving of roughly 350 MB per pull. DCO001 inspects the `FROM` instructions of the parsed Dockerfile and checks each image against a curated JSON mapping (`image_sizes.json`) of the most-pulled Docker Hub images to their slim and Alpine alternatives, together with compressed size deltas collected via the Docker Hub tags API. This rule is not auto-fixable by default because switching from a full image to a slim variant can break builds that depend on OS-level

tools (e.g., `gcc`, `curl`) removed in slim images. The fix is available via the `--force` flag after manual verification.

DCO002 - Uncombined `RUN` Layers. Each `RUN` instruction creates a new image layer. Consecutive `RUN` instructions that could be chained with `&&` produce unnecessary intermediate layers and reduce build cache efficiency. DCO002 identifies groups of two or more consecutive `RUN` instructions and reports them as a finding. The reported size saving is 0 MB because the overhead of extra layers cannot be measured statically; the benefit is improved layer hygiene and faster builds. The auto-fix merges each group into a single `RUN` instruction joined by `&& \`.

DCO003 - Development Dependencies in Production. Build tools such as `gcc`, `build-essential`, and `python3-dev` are required to compile native extensions but are unnecessary at runtime. Leaving them in the final image adds tens to hundreds of megabytes depending on the packages. DCO003 scans `RUN` instruction values in the final stage for known development package names listed in `dev_packages.json`, which contains per-package size estimates sourced from `apt-cache show` (Debian 12) and `apk info -s` (Alpine 3.19). Both `apt-get` and `apk` install patterns are supported. The auto-fix appends a cleanup command to the offending `RUN` instruction.

DCO004 - Missing `.dockerignore`. Without a `.dockerignore` file, the entire build context, including `.git` directories, test data, and local build artifacts, is sent to the Docker daemon on every build. DCO004 checks for the presence of a `.dockerignore` file in the build context directory. It scans for excludable directories (e.g., `.git`, `node_modules`, `.venv`) and sums their actual file sizes to produce a data-driven size estimate. The auto-fix detects the project language by probing for marker files (`go.mod` for Go, `requirements.txt` for Python, `package.json` for Node.js, `pom.xml` for Java) and generates a language-appropriate `.dockerignore` from a built-in template. If no language is detected, a generic template is used.

DCO005 - Missing Multi-Stage Build. Compiled languages such as Go, Rust, and Java produce large build environments that are not needed at runtime. Multi-stage builds allow the final image to contain only the compiled binary and its runtime dependencies. DCO005 detects single-stage Dockerfiles that use a compiled-language base image (Go, Rust, Maven, Gradle, Eclipse Temurin) and contain a compile command (`go build`, `mvn package`, etc.). The size saving is estimated by looking up the base image size in `image_sizes.json` and subtracting a 5 MB runtime baseline. No auto-fix is generated because restructuring a Dockerfile into multiple stages requires semantic understanding of the build process that cannot be safely automated; instead, a structured recommendation is provided.

DCO006 - Unpinned Base Image Tag. Using `FROM python` or `FROM python:latest` means the image resolves to a different layer set on each build, making builds non-reproducible and potentially introducing unintended dependency upgrades. DCO006 detects `FROM` instructions whose tag is absent, equal to `latest`, or pinned only to a major version (e.g., `python:3`). Tags with at least minor-version precision (e.g., `3.12`), distribution codenames (e.g., `bookworm`), and digest references are considered pinned. The auto-fix looks up the most recent stable pinned tag from the curated `image_sizes.json` dataset and substitutes it into the `FROM` line. Because tag pinning does not change image size, the reported size saving is 0 MB; the rule targets reproducibility rather than transfer cost.

3.4 Implementation Details

DCO is implemented in Python 3.10+ and depends on four runtime packages: `typer` (CLI framework), `rich` (terminal output), `dockerfile-parse` (Dockerfile AST), and `httpx` (Docker Hub API calls). `codecarbon` is available as an optional dependency (`pip install dco[energy]`) for users who wish to measure build-time energy using hardware performance counters. The rule registry uses a `@register` class decorator that instantiates each rule class and appends it to a module-level list; `pkgutil.iter_modules` auto-discovers all modules in `dco/rules/` without requiring manual registration. Rules communicate with the fix engine exclusively through the `FixAction` dataclass, which specifies an `action_type` string, a `(start_line, end_line)` tuple (0-indexed, inclusive), and `new_content`. The fixer validates the output Dockerfile by re-parsing it with `dockerfile-parse` and checking that at least one `FROM` instruction is present before writing to disk.

3.5 Validation and Reproducibility

All analysis logic is implemented as pure functions over strings and dataclasses, with no dependency on a running Docker daemon. The project includes 139 unit tests across detection rules, the fixer engine, the carbon estimator, and CLI integration. Test fixtures in `tests/fixtures/` provide sample Dockerfiles that cover each rule's detection and fix logic. The carbon model constants are defined in `dco/config.py`, making the model transparent and auditable. A `--no-dockerhub` flag disables all network calls for fully offline and reproducible CI runs.

For end-to-end validation, we built an automated pipeline that clones ten real-world repositories, replaces their Dockerfiles with intentionally un-optimized versions, runs DCO's analyze and fix commands, and builds the Docker images before and after optimization. Build energy is measured via CodeCarbon [9] using hardware power counters. Docker cache is fully pruned between builds to ensure fair comparison. The full validation procedure and results are described in section 4.

4 EVALUATION

4.1 Evaluation Setup

We evaluate DCO against ten real-world GitHub repositories spanning five programming languages: Python, Node.js, Java, Go, and PHP. For each repository we create an intentionally un-optimized Dockerfile that triggers a known set of DCO rules. The validation pipeline then measures the impact of DCO's fixes on image size, build time, and energy consumption.

The validation works as follows:

(1) Each repository is shallow-cloned (`git clone --depth 1`).
(2) An un-optimized Dockerfile from our test set replaces the original. For repositories where DCO004 should trigger, the `.dockerignore` is temporarily removed.

(3) `dco analyze` detects findings, then `dco fix` applies all safe fixes (with `-force` disabled for repositories where the base image swap would break the build).
(4) Docker cache is fully pruned (`docker system prune -a -f`) before each build to ensure fair comparison.
(5) Both the BEFORE (un-optimized) and AFTER (optimized) images are built with `-no-cache -pull` flags.
(6) CodeCarbon [9] measures build energy (kWh) and CO_2 (g) using hardware power monitoring (`powermetrics` on macOS).
(7) Image sizes and build times are recorded. Original files are restored after each repository.

All measurements were performed on a MacBook Pro (Mac16,1) with an Apple M4 Pro chip (12-core CPU, 16-core GPU) and 24 GB of unified memory, running macOS 26 with Docker Desktop. CodeCarbon used `powermetrics` for CPU power readings, run with `sudo` to access hardware counters. Docker Desktop was configured with default resource limits.

4.2 Results and Analysis

Table 2 shows the validation results for all ten repositories, sorted by size saved. In the table, "B." and "A." stand for Before (un-optimized) and After (optimized). "Rules" lists the DCO rule numbers that triggered (e.g., 001 = DCO001). Size is the compressed Docker image size in MB, time is the build duration in seconds, and CO_2 is the build emission measured by CodeCarbon in grams. All ten BEFORE and AFTER builds succeeded.

The results show a clear pattern: DCO001 (base image swap) drives the largest savings. The four repositories where DCO001 triggered and was applied (flask-sample-app, docker-curriculum, umami, directus) show size reductions between 308 and 355 MB. These repositories switched from full distribution images (e.g., `python:3.12`, `node:22.15`) to their slim variants.

For repositories where only DCO002 (combine RUNs) and DCO006 (pin tags) apply, the size difference is minimal or zero. This is expected: combining RUN layers improves build cache behavior but does not reduce compressed image size, and pinning a tag does not change the image content.

Gitea triggers five rules including DCO005 (missing multi-stage build), but DCO001's base image swap was disabled for this repository because switching from Debian to Alpine would break the `apt-get` commands in the Dockerfile. This demonstrates a real-world limitation: the most impactful fix cannot always be safely auto-applied.

Spring-boot-hello and appwrite show zero size reduction because their triggered rules (DCO002, DCO005, DCO006) do not affect compressed image size. DCO005 is detect-only (no auto-fix), and DCO002/DCO006 target build hygiene and reproducibility rather than size.

4.3 Energy and CO_2 Measurements

CodeCarbon recorded build energy for all ten repositories. The CO_2 values in Table 2 represent the emissions from a single Docker build on the test machine. For repositories with large size reductions, the AFTER build consistently uses less energy: flask-sample-app drops from 0.03 g to 0.02 g, and docker-curriculum drops from 0.02 g to near zero. The smaller optimized image requires less time to process, which translates to lower CPU energy consumption during the build.

These per-build CO_2 values are small in absolute terms because they measure a single local build. The larger sustainability impact comes from the network transfer savings estimated by Equation 1: a 355 MB reduction on an image pulled 100,000 times per month at the world-average grid intensity saves approximately 20 kg of CO_2 per month.

4.4 Rule Coverage

All six DCO rules are triggered at least twice across the test set:

- **DCO001** (5 repos): flask-sample-app, docker-curriculum, umami, directus, gitea
- **DCO002** (9 repos): all except docker-curriculum
- **DCO003** (3 repos): flask-sample-app, umami, directus
- **DCO004** (4 repos): getting-started-todo-app, spring-boot-hello, gitea, appwrite
- **DCO005** (2 repos): gitea, spring-boot-hello
- **DCO006** (7 repos): all except flask-sample-app, umami, directus

5 DISCUSSION

5.1 Interpretation of Results

The evaluation in section 4 shows that DCO001, the base image swap rule, is the primary driver of image-size reduction. The four repositories where DCO001 triggered and was applied show savings between 308 and 355 MB per image. By contrast, rules that target layer hygiene (DCO002) and reproducibility (DCO006) produce no measurable size reduction but improve build maintainability.

This result aligns with expectations: the difference between a full distribution image and its slim variant is measured in hundreds of megabytes, while combining RUN layers or pinning tags does not change the compressed image content. The practical implication is that developers seeking the largest carbon reduction should prioritize base image selection above all other optimizations.

5.2 The Force Flag Trade-off

DCO001 is not auto-fixable by default because switching from a full image to a slim variant can break builds that depend on OS-level tools removed in slim images. The gitea repository illustrates this: it uses `apt-get` to install build tools, so switching to an Alpine-based image would break the build entirely. We disabled `-force` for gitea, which meant DCO001 was detected but not applied, resulting in minimal size savings.

This trade-off is intentional. A tool that breaks builds to save image size would not be adopted. By requiring explicit opt-in via `-force`, DCO ensures that the base image swap only happens when the developer has verified compatibility.

5.3 Energy Measurement Observations

CodeCarbon's per-build energy measurements show a consistent pattern: optimized builds use less energy when the image size decreases. For flask-sample-app, the BEFORE build consumes 0.03 g

Table 2: DCO validation results across ten real-world repositories.

Repo	Lang	Rules	Size B. (MB)	Size A. (MB)	Saved (MB)	Time B. (s)	Time A. (s)	CO_2 B. (g)	CO_2 A. (g)
flask-sample-app	Py	001,002,003	403.8	48.4	355.4	49.9	39.1	0.03	0.02
docker-curriculum	Py	001,006	389.4	46.5	342.9	44.4	11.6	0.02	0.00
umami	Node	001,002,003	600.6	286.9	313.7	140.8	113.8	0.08	0.07
directus	Node	001,002,003	561.1	252.6	308.5	190.4	155.1	0.11	0.10
outline	Node	002,006	237.6	222.7	15.0	83.9	83.3	0.05	0.05
gitea	Go	001,002,004,005,006	1610.8	1602.4	8.4	203.1	239.3	0.16	0.16
getting-started	Node	002,004,006	409.4	402.4	7.0	125.2	127.4	0.10	0.09
etherpad	Node	002,006	107.2	106.7	0.5	48.1	45.3	0.03	0.02
spring-boot-hello	Java	002,004,005,006	225.8	225.8	0.0	45.6	47.7	0.02	0.02
appwrite	PHP	002,004,006	491.8	491.8	0.0	62.1	63.0	0.02	0.02

CO_2 while the AFTER build consumes 0.02 g, a 33% reduction that mirrors the 88% size reduction.

However, the per-build energy values are small because they measure a single local build on a power-efficient Apple Silicon laptop. The larger impact is at the network transfer level: every pull of the smaller image transfers fewer bytes, and at scale, this accumulates to meaningful energy savings. DCO's carbon model captures this by parameterizing the estimate with pull frequency and grid intensity.

5.4 Comparison with Existing Tools

Existing Dockerfile analysis tools such as hadolint [3] and dockle [6] focus on shell-scripting best practices and CIS benchmark compliance, respectively. Neither estimates carbon cost, neither produces a fixed Dockerfile, and neither provides machine-readable output (JSON/CSV) calibrated to deployment context. DCO fills this gap by coupling detection with estimation and remediation. Trivy [1] focuses on vulnerabilities and misconfigurations; its rule set is orthogonal to DCO's sustainability-focused patterns, making the two tools complementary.

5.5 Limitations

DCO's carbon estimates carry several sources of uncertainty. First, the Aslan et al. network intensity figure is extrapolated from 2015 data; the true 2026 intensity may differ if the halving trend has accelerated or stalled. Second, the image-size data in `image_sizes.json` is collected at a single point in time and may not reflect the latest image releases. A periodic refresh mechanism would reduce staleness risk. Third, DCO005 (missing multi-stage build) provides only a recommendation and cannot be auto-fixed, despite offering the largest potential savings for compiled-language images. Fourth, the tool estimates only transfer energy by default. Build-time energy is available through the optional CodeCarbon integration but is not factored into the standard output. Fifth, some size estimates are data-driven (DCO001 from `image_sizes.json`, DCO003 from `dev_packages.json`, DCO004 from directory scanning) while DCO002 reports 0 MB because layer overhead cannot be measured statically. Users should interpret DCO002 findings as build hygiene improvements rather than direct size savings.

6 FUTURE WORK

Several extensions would strengthen DCO's utility and accuracy.

Automated Multi-Stage Builds (DCO005). Multi-stage build restructuring is currently recommend-only because it requires semantic understanding of the build process. For common patterns in Go, Rust, and Java, where the build/runtime boundary is well-defined (compile in a builder stage, copy the binary to a minimal runtime image), automated fixes are feasible and would extend auto-fix coverage to the largest potential savings category.

Dynamic Image-Size Data. The curated `image_sizes.json` dataset is collected at a single point in time and may grow stale as maintainers release new base-image versions. An existing script queries Docker Hub for the latest data, but running it is a manual step. A dedicated subcommand that refreshes the dataset automatically would address this freshness concern.

CI/CD Integration. A GitHub Actions integration (or equivalent for GitLab CI, Jenkins, etc.) would allow DCO to run as an automated pull-request check, surfacing sustainability regressions alongside security and correctness checks. A `--fail-threshold` option that exits with a non-zero code when estimated CO_2 exceeds a configurable limit would enable quality gates on container sustainability.

Broader Rule Coverage. Additional rules could target patterns not yet covered: using `--no-install-recommends` with `apt-get`, leveraging layer caching through instruction ordering (placing frequently changing instructions last), and detecting redundant `COPY` instructions that invalidate the build cache. The plug-in rule architecture, one Python file per rule auto-discovered via `pkgutil`, makes adding new rules straightforward with no changes to the core engine.

Larger-Scale Validation. Our current evaluation covers ten repositories. Evaluating DCO against a larger corpus of public Dockerfiles (e.g., from GitHub or Docker Hub official images) would quantify the prevalence of each anti-pattern in the wild and provide stronger statistical evidence for the size-saving estimates.

7 CONCLUSION

This paper presented the Dockerfile Carbon Optimizer (DCO), a CLI tool that detects energy-wasteful patterns in Dockerfiles, estimates their monthly carbon cost using the Aslan et al. network energy

model [2] parameterized by pull frequency and grid intensity [8], and automatically rewrites Dockerfiles to eliminate detected inefficiencies.

We validated DCO against ten real-world GitHub repositories across five programming languages. The results show that the base image swap rule (DCO001) produces the largest impact, with size reductions of up to 366 MB per image. Repositories where DCO001 triggered also showed measurable reductions in build energy, confirmed via CodeCarbon [9] hardware energy monitoring on Apple Silicon.

Key Takeaways.

(1) The combination of detection, carbon-cost estimation, and automated remediation in a single tool removes the friction that prevents developers from acting on sustainability findings. Existing Dockerfile linters provide diagnosis but not treatment.
(2) Base image selection is the single most impactful optimization. Switching from `python:3.12` (392 MB) to `python:3.12-slim` (41 MB) saves 350 MB per pull, which at 100,000 pulls/month translates to approximately 20 kg CO_2/month.
(3) Not all findings can be safely auto-fixed. DCO001 requires the `--force` flag because slim images may break builds, and DCO005 (multi-stage builds) requires manual restructuring. This trade-off between automation and safety is intentional.
(4) The plug-in rule architecture, where each rule is a single Python file auto-discovered at runtime via `pkgutil`, makes DCO straightforward to extend. Adding a new rule requires no changes to the core engine.

REFERENCES

[1] Aqua Security. 2019. Trivy: Vulnerability Scanner for Containers and other Artifacts. https://github.com/aquasecurity/trivy. (2019).
[2] Joshua Aslan, Kieren Mayers, Jonathan G. Koomey, and Chris France. 2018. Electricity Intensity of Internet Data Transmission: Untangling the Estimates. *Journal of Industrial Ecology* 22, 4 (2018), 785–798. https://doi.org/10.1111/jiec.12630
[3] Lukas Brünink et al. 2016. Haskell Dockerfile Linter. https://github.com/hadolint/hadolint. (2016).
[4] CarbonFootprint.com. 2024. International Electricity Factors. https://www.carbonfootprint.com/international_electricity_factors.html. (2024).
[5] Climatiq. 2024. Climatiq: Carbon Emission Factors. https://www.climatiq.io/data/source/iea. (2024).
[6] Goodwith Inc. 2019. Dockle: Container Image Linter for Security. https://github.com/goodwithtech/dockle. (2019).
[7] Gaël Guénnebaud. 2024. Energy consumption of data transfer: Intensity indicators versus absolute estimates. *Sustainable Computing: Informatics and Systems* (2024).
[8] International Energy Agency. 2023. IEA Emission Factors 2023. https://www.iea.org/data-and-statistics/data-product/emissions-factors-2023. (2023). Accessed: 2025.
[9] Kadan Lottick, Silvia Susai, Sorelle A. Friedler, and Jonathan Wilson. 2019. CodeCarbon: Estimate and Track Carbon Emissions from Machine Learning Computing. https://github.com/mlco2/codecarbon. (2019).
[10] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. 2020. Recalibrating global data center energy-use estimates. *Science* 367, 6481 (2020), 984–986. https://doi.org/10.1126/science.aba3758
[11] Red Hat. 2023. dockerfile-parse: Python library for parsing Dockerfiles. https://github.com/containerbuildsystem/dockerfile-parse. (2023).

A UNIT TEST RESULTS

DCO includes 139 unit tests covering all detection rules, the fixer engine, the carbon estimator, and CLI integration. Table 3 summarizes the test distribution by module.

Table 3: Unit test distribution across DCO modules.

Test module	Tests
`test_rules/test_base_image` (DCO001)	15
`test_rules/test_run_layers` (DCO002)	8
`test_rules/test_dev_deps` (DCO003)	15
`test_rules/test_dockerignore` (DCO004)	9
`test_rules/test_multistage` (DCO005)	8
`test_rules/test_pinning` (DCO006)	18
`test_carbon/test_estimator`	13
`test_carbon/test_build`	3
`test_carbon/test_pull_frequency`	7
`test_parser`	14
`test_fixer`	16
`test_cli`	13
Total	**139**

All 139 tests pass on both Windows (Python 3.10) and macOS (Python 3.12). Two tests that depend on Docker Hub network access are skipped in offline mode.

B SOURCE CODE

The full source code for DCO is available at:
https://github.com/cs4575-g228/cs4575-g228-implementation