GCI: Wasteful Github CI Analyzer

CS4575 Sustainable Software Engineering

Maja Bińkowska (6509029) Priyansh Rajusth (6443621) Erkin Basol (5722330)
Jayran Duggins (5757312) Nicolas Loaiza Atehortua (6338771)

TU Delft, Delft, Netherlands

Abstract

Continuous Integration pipelines waste significant computational resources through unnecessary workflow executions, yet no widely available tool automatically identifies or quantifies this waste. We present GCI (Green CI Analyzer), a web application that heuristically classifies GitHub Actions waste across five categories: test flakiness, zombie scheduled workflows, external dependency failures, inefficient triggers, and workflow dependency cascade failures. GCI estimates energy consumption, CO_2 emissions, and financial cost using a carbon intensity model weighted across the 11 confirmed GitHub Actions Azure regions. Evaluated on five open-source repositories encompassing 750 manually labeled runs, GCI identified flaky jobs, classified zombie workflows with chronic failures, and traced dependent workflow relationships, as validated against workflow runs manually. Analysis with trigger and external dependency heuristics revealed that context sensitivity remains the primary challenge for future improvement. GCI is openly available at our repository.

1 Problem Statement

The growing adoption of Continuous Integration (CI) pipelines has introduced a largely overlooked dimension of technical debt: computational and energy waste from runs that consume real resources without producing any diagnostic signal. Every time a GitHub Actions workflow fires unnecessarily, triggered by a documentation-only commit, a flaky test, or a scheduled job on a long-deprecated runner, it consumes CPU time, and by extension, energy and money.

The scale of this problem is significant. Estimates place the carbon footprint of the GitHub Actions ecosystem at 456.9 $MtCO_2e$ for 2024 alone [16], while Huang and Lin found that inefficient CI triggering accounts for 79-83% of all computational waste across studied repositories, and that a single zombie scheduled workflow can silently waste over 757 days of compute time [8]. Despite this, CI service providers do not surface energy costs to developers, and no widely available tool automatically classifies the root causes of waste and estimates their environmental impact.

Existing work addresses related but narrower problems. Vassallo et al. detect structural pipeline misconfigurations in YAML files [22], but cannot identify waste patterns that only manifest in runtime execution history. Platform-native analytics report failure rates but do not classify failures by cause or estimate their energy cost.

We present **GCI**, a web application that addresses this gap. Given a repository URL, GCI retrieves GitHub Actions workflow run data via the REST API, classifies runs across five waste categories grounded in the taxonomy of Huang and Lin [8], estimates per-category energy consumption and CO_2 emissions, and presents findings through an interactive dashboard with prioritised recommendations. By making the hidden costs of CI waste visible and actionable, GCI supports both the environmental and economic dimensions of sustainable software engineering.

1.1 Research Questions

This work investigates how CI waste manifests in open-source repositories, how it can be measured using available API metadata, and what its estimated energy and carbon cost is:

RQ1 Which sources of CI waste are detectable using GitHub REST API workflow data (including metadata and logs)?

RQ2 How accurately do the GCI analyzers classify waste categories as evaluated through manual inspection of a set of workflow runs?

RQ3 What is the estimated energy and CO_2 cost of

the detected waste across the studied repositories?

1.2 Contributions

We make the following contributions:

1. **GCI**, a web application that automatically detects and quantifies CI waste in GitHub Actions repositories using five heuristic analyzers, each grounded in the empirical CI literature.
2. **An energy model** that estimates per-run energy consumption, CO_2 emissions, and financial cost. Carbon intensity is weighted across the 11 confirmed GitHub Actions Azure regions using the CO_2e data from the Cloud Carbon Footprint [2], an open-source tool that estimates energy consumption and CO_2e emissions across cloud providers region-specific grid emission factors, which draws on data from the U.S. Environmental Protection Agency (EPA) [19] and the European Environment Agency (EEA) [3].
3. **An evaluation** of GCI on open-source repositories, demonstrating the prevalence of each waste category and quantifying the potential energy savings from recommended fixes.

2 Background

2.1 Sustainable Software Engineering

Sustainable Software Engineering (SSE) focuses on designing, developing, and operating software systems in a way that minimizes negative environmental, economic, and societal impacts over time. The Karlskrona Manifesto proposed by Becker et al. establishes a widely adopted five-dimensional sustainability framework encompassing environmental, economic, technical, individual, and social concerns [1]

GCI connects to each dimension: it estimates energy consumption and CO_2 emissions of wasteful runs (environmental); quantifies the compute time lost to wasteful runs that could otherwise serve productive executions (economic); detects misconfigured workflows as a form of technical debt (technical); reduces flakiness-induced developer frustration by identifying unreliable pipelines (individual); and lowers the collective resource burden on open-source communities (social).

2.2 GitHub Actions and CI Pipeline Model

CI is a widely adopted software engineering practice in which code changes are automatically built, tested, and validated through automated pipelines [4]. By integrating changes frequently, CI provides rapid feedback on code correctness, though this comes at a computational cost: every push or pull request can trigger a full pipeline execution.

GitHub Actions is a CI/CD platform that executes workflows defined in YAML configuration files stored in a repository's `.github/workflows/` directory. A *workflow* is a pipeline definition consisting of one or more *jobs*, each executed on a virtual machine called a *runner*. A workflow is initiated by a trigger event, such as `push`, `pull_request`, `schedule`, or `workflow_run`, producing a *workflow run*. Each run is assigned a *run attempt* number, incremented on every manual or automatic retry, and is associated with a *head SHA* identifying the commit at the time of execution.

This execution model produces rich metadata per run, including status, duration, trigger event, run attempt count, workflow run ID, and head SHA. GCI leverages this metadata to detect waste patterns and estimate their energy and carbon impact, as described in the methodology.

2.3 CI Waste Prior Work

Prior work has established that CI pipelines can introduce significant computational and energy waste due to unnecessary or redundant executions. Sedano et al. [17] define software development waste as effort that consumes resources without delivering value, a concept that directly applies to CI systems where runs may execute without providing new diagnostic information.

Recent empirical studies have investigated specific sources of CI inefficiency. Huang and Lin [8] analyze reruns in GitHub Actions workflows and show that a substantial portion of computational time is wasted on executions that do not change outcomes. Their findings further indicate that CI workflows, due to their high triggering frequency, are responsible for the majority of rerun-related waste. Similarly, Maipradit et al. [12] study repeated builds during code review and find that many rechecks are performed despite a low likelihood of outcome change, highlighting redundancy in CI execution.

Other work has focused on waste induced by changes that do not affect system behavior. Weeraddana et al. [23] show that a large fraction of CI build time is triggered by updates to unused dependencies, demonstrating that certain changes can initiate full pipeline executions without impacting functionality. In parallel, research on CI/CD pipeline configuration has identified structural inefficiencies, such as misconfigured workflows and lack of filtering mechanisms, which can lead to unnecessary

executions [22].

While these studies provide valuable insights into different sources of CI waste, they largely focus on individual aspects such as reruns, dependency updates, or configuration issues in isolation. In contrast, there is limited work on automatically detecting and categorizing multiple sources of CI waste using runtime execution data, or on quantifying their environmental impact. This gap motivates the need for tools such as GCI, which aim to systematically identify, classify, and estimate the cost of CI inefficiencies across multiple dimensions.

2.4 Identified CI Waste Sources

Huang and Lin [8] identify five recurring root causes of computational waste in GitHub Actions pipelines, which we adopt in this study.

Non-deterministic test flakiness accounts for 32% of workflow-level and 42% of job-level flakiness. Failures stem from transient issues , timeouts, unstable connections, non-deterministic test ordering , rather than code defects. Maipradit et al. found that 55% of code reviews trigger a recheck after a failing build, but the outcome changed in only 42% of cases, meaning the majority of reruns are wasteful [12].

Zombie scheduled workflows The scheduled (fully automated) workflows are responsible for 54.8% of all reruns. When scheduled workflows silently rely on deprecated runner configurations, they can fail continuously for months undetected. We call scheduled workflows that fail continuously and repeatedly over a period of time 'Zombies'. Huang and Lin document a single case that wasted 757 days of waiting time from one workflow [8].

External dependency and service instability, including bugs in third-party actions and registry or cloud provider outages, accounts for 21% of workflow flakiness and compounds waste through cascading reruns [8].

Inefficient CI triggering for minor changes , running full test suites on documentation edits or unused dependency updates , is responsible for 79–83% of all computational waste across CI workflows [8].

Workflow dependency cascade failures occur when child workflows triggered via `workflow_run` are doomed from the start because their parent was already cancelled or failed, wasting the full execution time of the child run.

2.5 Energy Estimation for CI Pipelines

Directly measuring the exact energy consumption of CI pipelines on GitHub Actions is difficult because runs execute on managed cloud infrastructure without per-run energy telemetry. As a result, prior work adopts estimation-based methods that approximate consumption from runtime and hardware assumptions [11, 13]. The core idea is that energy use scales with power and execution time, with an additional overhead factor for data-center operations represented by Power Usage Effectiveness (PUE). PUE is defined as the ratio of total facility energy to IT equipment energy; values closer to 1 indicate more efficient facilities [10, 15].

Following this literature, we estimate both environmental and economic CI waste. Environmental impact is quantified via estimated energy and carbon emissions, while economic impact is quantified via billed runner cost.

2.6 Repository Selection and GitHub Data Collection

GitHub Actions provides a rich source of publicly accessible CI/CD data through its REST API. This data includes information about workflow runs, jobs, commits, timestamps, and execution outcomes.

These metadata enable the reconstruction of CI pipeline behavior over time, allowing researchers to analyze execution patterns, failure rates, and triggering events across repositories.

Available information includes workflow run status, execution duration, run attempts, associated commits, and changed files, as well as optional job-level details for deeper inspection.

However, this data has several limitations. GitHub does not expose direct energy consumption metrics, requiring indirect estimation methods. Additionally, access to detailed execution logs, which are required for certain failure classes like External Dependency Failures, may be restricted or require additional API calls, and rate limits constrain the volume of data that can be retrieved.

These constraints motivate the need for heuristic-based approaches to identify inefficiencies and estimate their environmental impact.

3 Methodology

3.1 Approach

The objective of this study is to design, implement and evaluate GCI, a tool that automatically heuristically detects and quantifies CI waste in GitHub Actions repositories. Our goal is to analyze GitHub Actions workflow run histories to identify and estimate the energy and carbon cost of CI waste in the context of publicly accessible GitHub repositories. The study evaluates GCI against the five waste categories identified by Huang and Lin [8], assessing both the detectability of each category using Github Actions

API data (metadata and logs) and the precision of the associated energy estimates.

3.2 System Architecture

GCI is implemented as a two-tier web application. The backend is a Python Flask server exposing three endpoints: a streaming analysis endpoint (`POST /api/analyze/stream`), a failure diagnosis endpoint (`POST /api/diagnose`), and a health check (`GET /api/health`). This separation of concerns between data retrieval, analysis, and presentation keeps the system modular and allows each analyzer to be tested and extended independently.

Analysis results are delivered to the client incrementally via Server-Sent Events (SSE), which is well-suited to this use case: because GitHub API pagination and five sequential analyzers can take tens of seconds on large repositories, SSE allows the frontend to render findings progressively rather than blocking on a single long-lived HTTP response. A `GitHubClient` module wraps the GitHub Actions REST API, handling authentication, pagination, ETag-based caching, and rate-limit back-off - isolating all API concerns in one place and making the analyzers independent of the underlying HTTP mechanics.

The five waste analyzers: `FlakinessAnalyzer`, `ZombieWorkflowAnalyzer`, `ExternalDepsAnalyzer`, `InefficientTriggerAnalyzer`, and `WorkflowDependencyAnalyzer` run sequentially over the fetched workflow run data, each producing a structured findings report. An `energy` module then estimates per-run energy consumption, CO_2 emissions, and financial cost using runner-type detection and IP-weighted carbon intensity across the 11 confirmed GitHub Actions Azure regions. Keeping energy estimation as a separate post-processing step means the analyzers remain focused on waste classification, and the energy model can be updated independently as emission factors change.

The frontend is a single-page React 18 application built with Vite, that consumes the SSE stream and renders results through a stepped dashboard using Chart.js bar and doughnut visualisations. Deploying the frontend as a static SPA with a lightweight API backend minimises operational overhead, requiring no persistent database or background workers, the GitHub API itself serves as the data source.

3.3 Data Collection

GCI retrieves workflow run data from the GitHub Actions REST API collecting up to latest 300 runs filtered by a date range per repository across three paginated requests of 100 runs each. This limit reflects a deliberate trade-off between API budget and analysis coverage: each run subsequently requires additional calls for job-level and commit-level data, so capping the run count keeps total API consumption within the authenticated budget of 5,000 calls per hour.

For each run, we extract its identifier, workflow, trigger event, conclusion, retry count, commit SHA and message, and timestamps for creation, start, and completion. Where individual analyzers require it, two additional data sources are consulted: changed file paths per commit, used by the inefficient trigger analyzer, and job-level execution data, used by the flakiness and dependency analyzers.

Rate limit pressure is managed through two mechanisms. First, all cacheable requests use ETag-based conditional HTTP requests, so repeated calls for unchanged resources return HTTP 304 and do not count against the hourly quota. Second, the client monitors the remaining budget header and pauses execution when fewer than five calls remain, resuming after the reset window. Together, these ensure GCI degrades gracefully on large repositories rather than failing mid-analysis.

3.4 Waste Detection Metrics

GCI implements five heuristic analyzers, each targeting one waste category identified by Huang and Lin [8].

Flakiness. Prior research shows many CI failures are non-deterministic and can pass on retry without code changes. GCI detects flakiness at the job level by comparing outcomes across rerun attempts within the same workflow run, excluding issues from PR state changes like labels or reviews.

GCI identifies runs where `run_attempt` is greater than 1 and fetches job data for all attempts using `/runs/id/attempts/n/jobs`, grouping them by `(run_id, job_name)`. A job is flaky if it shows both failures and successes across different attempts of the same run.

GCI does not compare jobs across different run IDs on the same commit, since this would conflate real flakiness with PR state changes between separate runs. By comparing only attempts within a single run, GCI isolates true non-deterministic failures.

To measure waste, GCI counts failed job instances where `run_attempt` is greater than 1 within flaky groups. Runs triggered by `schedule` are excluded to avoid overlap with zombie analysis, and `action_required` jobs are excluded as approval gates. The summary reports distinct flaky jobs and total failures, showing whether flakiness concentrates in one job or spreads across many.

Zombie Scheduled Workflows. In GCI a scheduled workflow is classified as a zombie if more than 80% of its runs within the study window conclude as `failure` and it has at least three scheduled runs. This 80% threshold separates workflows that are reliably broken and chronically waste resources on an autopilot schedule from those with only occasional failures. The minimum three-run requirement balances statistical significance with early detection: too few runs risk false positives from noise, while too many delay the signal of emerging problems.GCI additionally detects consecutive failure streaks of five or more runs to identify workflows entering acute failure periods, even if overall failure rates remain below the zombie threshold; this complementary approach captures both chronic infrastructure decay and sudden breakage. The analyzer further scans workflow YAML files for deprecated runner labels e.g. `ubuntu-18.04`, `macos-11` and outdated action versions e.g. `actions/checkout@v1`, targeting the specific root cause identified in prior research: outdated infrastructure and EOL dependencies are a leading driver of zombie workflows, enabling operators to distinguish infrastructure-related failures from code defects and prioritize remediation toward the infrastructure fixes that will unlock the most corrections. GCI applies a recency filter: if the most recent execution of a high-failure workflow is successful, it is excluded from the zombie classification to account for recent fixes.

External Dependency Failures. Failures caused by third-party service instability are identified exclusively through log-level pattern analysis of CI runs, with metadata-based heuristics used only to prioritize investigation. Candidate runs are first ranked using signals such as early-death transients (runs that fail in under 40% of the median successful duration but later succeed on the same commit), temporal clusters (multiple distinct workflows failing within a short time window, indicating a likely outage), and failures in third-party actions or setup/install steps. Job logs from prioritized runs are then analyzed using pattern matching over a curated set of error signatures, including network failures, DNS resolution issues, HTTP 5xx and 429 responses, SSL errors, and registry-specific failures across ecosystems such as npm, pip, Maven, and Docker. Only runs with at least one matched error pattern in the logs are classified as external dependency failures, so that results reflect evidence of third-party instability rather than prior heuristic suspicion.

Inefficient Triggers. CI/CD systems rely on automated triggers such as pushes and pull requests to initiate workflow executions. Prior work shows these are a major source of waste, accounting for 79-83% of rerun-related computational overhead [8]. From a software engineering perspective, such waste corresponds to computational effort that does not produce meaningful value [17].

We define an *inefficient trigger* as a `push` or `pull_request` run with conclusion `success` or `failure` that executes a *heavy* workflow even though all modified files are classified as non-functional. In the implementation, non-functional changes include documentation directories, documentation file types (e.g., `.md`, `.rst`, `.adoc`), common image formats, and metadata-style files such as `README`, `LICENSE`, and `CHANGELOG`. Prior work similarly shows that non-functional changes can still trigger unnecessary CI activity, for example when updates to unused dependencies initiate pipelines without affecting system behavior [23].

We classify workflows as *heavy* or *light* using keyword-based matching over the workflow name and path. Heavy workflows contain terms such as `build`, `test`, `deploy`, `docker`, or `release`, following common CI/CD naming conventions [5, 6]. Workflows lacking path-based filtering (`paths` or `paths-ignore`) are treated as stronger evidence, although path filtering only affects the explanation attached to a flag, not whether the run is flagged. While these heuristics do not guarantee perfect accuracy, they provide a practical and scalable approximation for identifying likely inefficient triggers in CI/CD pipelines.

Workflow Dependency Cascade Failures. Workflow dependency cascade failures occur when a workflow is triggered by the completion of another workflow's execution through the event `workflow_run`, typically using the output of the parent workflow [8].

This phenomenon is important because even when a workflow run fails, its dependent workflow may still execute, despite being highly likely to fail as well. Detecting such dependencies enables developers to introduce conditions that control job execution and eliminate unnecessary resource consumption.

To identify dependent workflows, GCI parses all workflow YAML files to detect `workflow_run` dependencies. Specifically, it checks for the **on: workflow_run:** trigger. For each child workflow run, the corresponding parent run is identified by matching the `head_sha`, which represents the commit associated with the workflow execution. Since workflows triggered via `workflow_run` are executed in response to the completion of another workflow on the same commit, the `head_sha` serves as a key to associate parent and child runs. A child run is classified as a cascade failure if it concludes with `failure`

or `cancelled`, and its parent run also concludes with `failure` or `cancelled`, or if no parent run is found.

3.5 Energy-Carbon-Cost Estimation

For each flagged run, GCI estimates energy consumption as:

$$E_{\text{kWh}} = \frac{P_{\text{runner}} \times t_{\text{run}} \times \text{PUE}}{3{,}600{,}000} \tag{1}$$

where t_{run} is wall-clock runtime in seconds, P_{runner} is runner power draw in watts, and PUE accounts for data-center overhead.

Runner power is estimated using an empirical baseline of 3.57 W for GitHub-hosted Linux runners (AMD EPYC 7763), based on Green Coding measurements [13]. An average 1.17 PUE is used using Microsoft's published data-center efficiency information [15]. This value represents an approximate CPU-level baseline and likely underestimates full-system energy consumption.

To estimate carbon emissions, we approximate the execution region of workflows. GitHub Actions IP ranges are obtained via the GitHub `/meta` endpoint and mapped to Microsoft Azure regions using Azure's published IP range dataset [14]. As individual runs cannot be linked to specific regions, we assume a uniform distribution across all identified locations.

Carbon emissions are calculated as [24] following the GHG scope 2 protocol:

$$C = E \times I \tag{2}$$

where I is the carbon intensity (gCO_2/kWh). Region-specific intensity values are obtained from [2], which draws data from the EPA [19] and EEA [3]. We use an average intensity across all regions, and report uncertainty as a range based on minimum and maximum values.

To improve actionability and translate CI waste into a metric that is directly meaningful to practitioners, GCI also estimates monetary waste by multiplying the wall-clock runtime of each flagged run by the GitHub Actions per-minute price for the corresponding detected runner category [7]:

$$\text{Cost}_{\text{USD}} = \sum_{r \in \mathcal{R}} \left(\frac{t_r^{\text{run}}}{60} \right) \times p_r \tag{3}$$

where t_r^{run} is the wall-clock runtime of flagged run r in seconds and p_r is the per-minute price assigned to the detected runner category.

3.6 Impact Comparisons

To make estimated energy use and carbon emissions easier to interpret, GCI converts them into a small set of illustrative everyday equivalents. These are intended as communication aids rather than direct measurements [21].

Carbon-based comparisons include smartphone charges, passenger-car travel, video streaming, gallons of gasoline burned, and days of household electricity emissions. The implementation uses EPA factors for smartphone charging, passenger-vehicle travel, gasoline, and home electricity use, while streaming is based on the IEA estimate of 36 g CO_2 per hour. Car distance is displayed in kilometers for readability, but is computed from the EPA grams-per-mile factor [9, 20, 21].

For energy-based comparisons, GCI reports how long the same amount of electricity could power an LED bulb. It uses a representative bulb power of 8.25 W, taken as the median of DOE example efficient bulb ratings of 5.5 W and 11 W [18].

3.7 Experiment Design

GCI is evaluated on five publicly accessible GitHub repositories by comparing the waste classifications produced by each heuristic analyzer against a manually constructed ground truth. For each repository, the latest 150 workflow runs between 01.01.2026 and 01.04.2026 are used, yielding a total evaluation corpus of 750 labelled runs. Each run is evaluated by all five analyzers independently and may be flagged under multiple waste categories. Ground truth labels are assigned per analyzer, based on manual inspection of the run metadata, commit history, workflow configuration, and execution logs where necessary.

3.8 Repository Selection

For evaluation, we have chosen 5 repositories that motivate the use of our tool to stakeholders. We identified the following:

vividus-framework/vividus - We selected this repository because it is explicitly referenced in prior work as an example of job-rerun flakiness in `gradle.yml` [8]. This made it a strong literature-grounded repository to evaluate GCI before running any analysis.

The repository is also practical for our study: it has active, test-heavy CI workflows where unstable reruns and environment-related failures can directly affect developer feedback speed and maintenance cost. For stakeholders (maintainers and contributors), identifying avoidable CI waste is therefore immediately useful.

Our pre-analysis hypothesis was that this repository would show measurable rerun-related waste. We then validated this by comparing analyzer outputs with manual run-level inspection.

avelino/awesome-go - The `awesome-go` repository was selected as a case study for evaluating our tool due to its high popularity, large contributor base, and active development patterns. With approximately 169k stars, 13k forks, and over 2,000 contributors, it represents a highly collaborative and widely used project, making it a suitable environment for observing potential inefficiencies. The large number of contributors also implies a diversity of development practices and experience levels, which increases the likelihood of inconsistencies in commit behavior and workflow usage, potentially leading to wasteful executions such as redundant runs or unnecessary updates. However, because the repository is mature and widely maintained, we do not expect to observe a high frequency of issues such as flaky workflows or failures caused by unstable external dependencies, as such problems are more likely to be identified and resolved quickly in a project of this scale and visibility.

01-edu/public - We selected the `01-edu/public` repository because it is likely to exhibit several common CI-related waste patterns at a meaningful scale. With over 6,300 commits and more than 2,500 workflow runs, the repository provides sufficient activity to observe recurring inefficiencies rather than isolated incidents. It defines multiple GitHub Actions workflows-such as Prettier checks, link validation, ShellCheck, and Docker image builds-that are all triggered on pull request events. This configuration means that a single push to a PR branch can initiate several parallel workflow runs, increasing the risk of redundant or unnecessary executions, particularly when changes are small or iterative.

elastic/apm-agent-java - The `elastic/apm-agent-java` repository was selected to validate workflow dependency issues because it has been explicitly referenced in prior work as an example of cascading workflow failures triggered via the `workflow_run` event [8]. By selecting a repository already identified in the literature as exhibiting these characteristics, we provide a strong foundation for evaluating the effectiveness of the GCI tool in detecting workflow-related waste. Moreover, the repository contains 17 workflow configuration files, suggesting a relatively extensive CI/CD setup that increases the likelihood of redundancy, misconfiguration, or inactive workflows, making it a suitable candidate for further analysis.

deepjavalibrary/djl-serving - The `deepjavalibrary/djl-serving` repository was selected because it has been identified in existing research as a notable example of CI flakiness caused by external service dependencies [8]. The repository contains 26 workflow files, including scheduled workflows, dependent workflows, and workflows that rely on external cloud infrastructure such as AWS instances for LLM integration testing. This combination of scheduled reruns, interdependent workflows, and reliance on external services increases the likelihood of wasted executions, failures, or instability, particularly when external resources are slow, unavailable, or inconsistent. These characteristics make the repository a representative case for evaluating the GCI tool's ability to detect dependency-related waste in complex CI/CD environments.

4 Results

We evaluated GCI Wasteful-CI-Analyzer on 750 workflow runs across five repositories (150 each), and report manually validated results per analyzer in the subsections below.

4.1 Flakiness

The analyzer showed perfect agreement with manual labeling in the evaluated cases. In `vividus-framework/vividus`, all 18 flagged runs were confirmed as true positives. Among 39 inspected multi-attempt runs, 18 (46%) exhibited non-deterministic behavior, visible as job groups alternating between failure and success across retry attempts within the same run. Three repositories produced only true negatives. `deepjavalibrary/djl-serving/` yielded 5 confirmed true positives out of 7 inspected reruns (71%), again showing the same alternating pattern.

Notably, rerun counts in vividus reached up to 19 attempts for individual runs, suggesting that retries may be masking persistent root causes rather than resolving them.

4.2 Zombie Scheduled Workflows

The 80% failure threshold correctly identified `sagemaker-integration.yml` in `deepjavalibrary/djl-serving` which failed persistently due to an unresolved `ImportError`. Other workflows, such as `nightly.yml` and `serving-publish.yml` exhibited long consecutive failure streaks but did not exceed the aggregate failure threshold. These were therefore left unclassified, which aligns with the intended conservative definition.

All remaining repositories showed only occasional scheduled failures and were consistently classified as non-zombies. This indicates that the threshold effectively distinguishes chronic failure from transient instability.

4.3 External Dependency Failures

In `avelino/awesome-go`, the `ERR_REQUIRE_ESM` error caused by an incompatible ES module import was consistently and correctly flagged. In `01-edu/public`, all runs were correctly classified as true negatives.

In contrast, `vividus-framework/vividus` exhibited a high number of false negatives. Failures originating from Mobitru, SauceLabs, and BrowserStack were not detected, as their provider-specific error signatures are absent from the current pattern library. Conversely, in `elastic/apm-agent-java` and `deepjavalibrary/djl-serving`, false positives occurred because generic network error patterns matched within test execution output rather than true dependency failures.

These results highlight a key limitation: the analyzer is sensitive to both missing patterns and context-insensitive matching.

4.4 Inefficient Triggers

In `vividus-framework/vividus`, one true positive and one false positive were identified. The false positive occurred because the analyzer treated a release/tag push the same as a regular development push. In practice, release-related runs may still be intentional and necessary, even when the associated commit contains only non-functional files. In `avelino/awesome-go`, all 49 flagged runs were false positives because the repository's primary output is its `README`. While the analyzer correctly marked these runs as documentation-only, such changes are core to the project itself, meaning that CI triggered by `README` updates is not inherently inefficient but actually necessary for the repo's purposes. Similarly, in `01-edu/public`, 7 flagged runs were classified as false positives. Although triggered by `README` changes, these files are part of the project's core content and may be essential to their docker use cases. Some runs were also misclassified because a Docker-image sanitization workflow was treated as heavy due to the keyword `docker`. Therefore, labeling these runs as wasteful would be misleading.

Inefficient triggers were not detected in `elastic/apm-agent/java`. The repository appears to use well-configured CI workflows that our heuristic could not flag as wasteful. Notably, some workflows are triggered on a schedule and may run without new commits; however, scheduled runs are not considered by our trigger analysis. Additionally, many runs are initiated by automated actors (e.g., GitHub bots), where determining inefficiency requires deeper contextual analysis beyond commit contents.
In `deepjavalibrary/djl-serving`, scheduled runs repeatedly executed on an unchanged repository state. These were classified as true negatives under the current definition, but would likely be considered false negatives under a broader notion of trigger inefficiency.

Overall, the results demonstrate that trigger classification is highly dependent on repository context, which is not currently modeled.

4.5 Workflow Cascade Failures

Out of the five analyzed repositories, only two, `elastic/apm-agent-java` and `deepjavalibrary/djl-serving`, exhibited observable workflow dependency patterns. The remaining three repositories did not show any evidence of workflow dependencies.

In `elastic/apm-agent-java`, a total of 39 dependent workflow runs were identified. These runs were triggered on the same commit SHA as their corresponding parent workflows, confirming the existence of dependency relationships. Importantly, these dependencies did not result in cascade failures. Our tool, GCI, correctly identified these relationships, accurately classifying workflows such as `test-reporter.yml` and `docs-deploy` as dependent on upstream workflows based on their trigger configurations and shared execution context.

In `deepjavalibrary/djl-serving`, 47 dependent workflow runs were detected. All runs were associated with the workflow `Publish Job Success Metric to CloudWatch`, which is configured to execute upon the completion of any workflow on the main branch. These runs also did not result in cascade failures. GCI correctly recognized these dependent workflows by identifying their triggering patterns and associating them with the corresponding upstream workflow executions.

5 Discussion

Our findings indicate that CI waste detection is highly sensitive to repository context. While several analyzers performed well under expected conditions, errors emerged when workflow intent, dependency structure, or trigger semantics differed from heuristic assumptions. We discuss these patterns and their implications for improving robustness.

5.1 Flakiness

The analyzer achieves high precision due to a deliberate scoping decision: job outcomes are compared only across retry attempts within a single run ID, and never across separate runs on the same commit SHA. This prevents changes in pull request state

(e.g., labels or review assignments) from being misinterpreted as nondeterminism.

However, this design introduces a limitation. Environment-level flakiness that manifests across separate runs on the same commit remains undetected. Additionally, the extreme retry counts observed in vividus point to a distinct issue: workflows may eventually pass due to permissive retry configurations, masking underlying instability.

Detecting this behavior would require correlating rerun counts against repository-level baselines to identify statistically anomalous retry patterns.

5.2 Zombie Scheduled Workflows

GCI identifies zombie scheduled workflows using an 80% failure-rate threshold over recurring executions. In `deepjavalibrary/djl-serving`, it detected 44 scheduled workflows, most of which had recurring runs. The workflow `sagemaker-integration.yml` consistently failed due to an unresolved import error, exceeding the threshold and being correctly classified as a zombie workflow, highlighting a lack of maintenance.

In contrast, workflows such as `nightly.yml` and `serving-publish.yml` showed consecutive failure streaks but were not classified as zombies, as their overall failure rates remained below 80%. This demonstrates that GCI distinguishes persistent failures from intermittent ones based on aggregate behavior.

Other repositories showed only occasional failures, and GCI correctly classified all of them as non-zombie workflows, indicating accurate and balanced classification.

5.3 External Dependency Failures

Observed false positives in `elastic/apm-agent-java` and `deepjavalibrary/djl-serving` stem from a shared implementation issue: `_analyze_run_logs` concatenates the entire log into a single string before pattern matching. As a result, the analyzer lacks awareness of pipeline phases. For instance, a "connection reset by peer" occurring within a test's HTTP mock layer is indistinguishable from a genuine dependency resolution failure.

A straightforward improvement would be to segment logs by step and restrict pattern matching to setup and installation phases, leveraging step-level markers already present in GitHub Actions logs.

The false negatives in vividus represent a different challenge. Provider-specific errors from Mobitru, SauceLabs, and BrowserStack are not included in the current pattern library. Addressing this gap would require either substantial expansion of the pattern set or a more flexible approach, such as confidence-based aggregation of partial signals across multiple categories.

5.4 Inefficient Triggers

A key finding is that determining whether a CI run is inefficient is highly dependent on the repository's context and intended use. The current heuristic assumes that changes to files such as `README.md`, documentation, or images are non-functional. While this is reasonable for many traditional software projects, it does not generalize to repositories where such files are part of the core output or directly influence workflow behavior. This led to clear false positives in repositories such as `avelino/awesome-go` and `01-edu/public`.

The analyzer also struggles with workflows triggered by automated accounts. Bots such as Dependabot or GitHub Actions may trigger runs as part of maintenance or chained workflows. Without understanding this broader context, it is difficult to determine whether such runs are unnecessary. This limitation was particularly visible in `vividus-framework/vividus` and `elastic/apm-agent-java`.

In addition, the current implementation does not account for scheduled workflows. In practice, scheduled runs can repeatedly execute on the same commit without any new changes, producing no meaningful new result. These cases can also represent inefficient triggering, but are currently outside the scope of the analyzer. This was observed in repositories such as `deepjavalibrary/djl-serving`.

At a more technical level, the `_is_non_functional` function applies a fixed rule-based classification of file types, which does not capture repository-specific meaning. Similarly, the event filter considers only `push` and `pull_request` events without distinguishing between branch and tag-based triggers, treating them as equivalent despite their different purposes.

5.5 Workflow Cascade Failures

Of the five repositories analyzed, only `elastic/apm-agent-java` and `deepjavalibrary/djl-serving` exhibited observable workflow dependency patterns, each revealing a distinct limitation of our approach. In `elastic/apm-agent-java`, workflows such as `test-reporter.yml` are explicitly triggered upon completion of `main.yml`. An important insight emerged from this case: dependent workflows can succeed even when their parent workflow fails. This occurs because the dependency effectively operates at the job level; `test-reporter.yml` only requires the `Build and Test Windows` job to complete and

upload artifacts, regardless of whether other jobs in `main.yml` fail. This highlights a limitation of our current approach, which considers only the final conclusion of the parent workflow. A more robust method would trace dependencies between individual jobs and classify a scenario as a dependency cascade only when a downstream failure is causally linked to a specific upstream job failure. In `deepjavalibrary/djl-serving`, the `Publish Job Success Metric to CloudWatch` workflow is configured with a wildcard trigger (`workflows: ["*"]`), running on the completion of any workflow on the main branch. Although GCI initially identified 47 dependent runs, no consistent commit-level matches could be established with parent workflows. This is because the workflow only observes when other workflows finish, without consuming any of their outputs or artifacts. To address this, we refined GCI to detect wildcard-triggered workflows and exclude them from cascade failure analysis. This distinction underscores the importance of differentiating between true workflow dependencies and passive monitoring workflows when analyzing cascade behavior.

5.6 Cross-cutting Limitation

Bot-triggered runs appear across vividus, elastic, and djl-serving, yet the current analyzers treat them identically to human-triggered runs. In practice, events triggered by Dependabot or `github-actions[bot]` differ significantly in intent and behavior from developer-initiated changes.

The `actor` field, available via the GitHub API, provides a straightforward mechanism for identifying such cases. Incorporating this signal would enable differentiated handling, such as adjusted thresholds or selective exclusion, and would likely improve precision across multiple analyzers with minimal implementation cost.

6 Conclusion

This work presented GCI, a web application that automatically detects and quantifies CI waste in GitHub Actions repositories across five categories: test flakiness, zombie scheduled workflows, external dependency failures, inefficient triggers, and workflow dependency cascade failures. By combining heuristic analyzers with an energy-carbon-cost estimation model weighted across GitHub Actions' confirmed Azure regions, GCI makes the hidden environmental and economic costs of wasteful CI executions visible and actionable.

Looking ahead, we envision several directions for extending GCI into a more comprehensive sustainability instrument for CI/CD pipelines. Second, incorporating repository-specific context, such as learning which files constitute core project output versus auxiliary documentation, would substantially reduce false positives in the inefficient trigger analyzer, addressing the most prominent source of misclassification observed in our evaluation. Third, real-time integration as a GitHub App or Actions workflow would allow GCI to provide per-commit sustainability feedback directly within the developer workflow, shifting waste detection from retrospective analysis to proactive prevention. Finally, as cloud providers begin exposing more granular energy telemetry and region-level execution metadata, GCI's estimation model could be refined from statistical approximation toward empirical measurement, increasing both the accuracy and credibility of its environmental impact reporting. By continuing to surface the hidden costs of CI inefficiency, tools like GCI can help embed sustainability as a first-class engineering concern, encouraging development teams to treat computational waste with the same rigor they apply to code quality and system reliability, ultimately reducing the growing carbon footprint of modern software delivery infrastructure.

References

[1] Christoph Becker, Ruzanna Chitchyan, Leticia Duboc, Steve Easterbrook, Birgit Penzenstadler, Norbert Seyff, and Colin Venters. Sustainability design and software: The karlskrona manifesto. pages 467–476, 05 2015. doi:10.1109/ICSE.2015.179.

[2] Cloud Carbon Footprint. Cloud carbon footprint methodology, n.d. Accessed April 2, 2026. URL: https://www.cloudcarbonfootprint.org/docs/methodology/.

[3] European Environment Agency. Co2 emission intensity of electricity generation, 2024. Accessed: 2026-04-02. URL: https://www.eea.europa.eu/en/analysis/maps-and-charts/co2-emission-intensity-15.

[4] Martin Fowler. Continuous integration, 2024. URL: https://martinfowler.com/articles/continuousIntegration.html.

[5] GitHub. Github actions ci/cd best practices, 2024. Accessed: 2026-03-29. URL: https://github.com/github/awesome-copilot/blob/main/instructions/github-actions-ci-cd-best-practices.instructions.md.

[6] GitHub. Workflow syntax for github actions, 2024. Accessed: 2026-03-29. URL: https://docs.github.com/en/actions/reference/workflows-and-actions/workflow-syntax.

[7] GitHub. Github actions runner pricing. https://docs.github.com/en/billing/reference/actions-runner-pricing, 2026. Accessed: 2026-04-02.

[8] J. Huang and B. Lin. On the reruns of GitHub Actions workflows. *ACM Transactions on Software Engineering and Methodology*, February 2026. doi:10.1145/3795771.

[9] International Energy Agency. The carbon footprint of streaming video: Fact-checking the headlines, 2020. Accessed: 2026-04-02. URL: https://www.iea.org/commentaries/the-carbon-footprint-of-streaming-video-fact-checking-the-headlines.

[10] Paul Kirvan, Alexander S. Gillis, and Mark Fontecchio. What is pue (power usage effectiveness)?, June 2025. Accessed: 2026-03-30. URL: https://www.techtarget.com/searchdatacenter/definition/power-usage-effectiveness-PUE.

[11] Patricia Lago, Sedef Akinli Koçak, Ivica Crnkovic, and Birgit Penzenstadler. Framing sustainability as a property of software quality. In *Communications of the ACM*, volume 58, pages 70–78, 2015. doi:10.1145/2714560.

[12] Rungroj Maipradit, Dong Wang, Patanamon Thongtanunam, Raula Gaikovina Kula, Yasutaka Kamei, and Shane McIntosh. Repeated builds during code review: An empirical study of the OpenStack community. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM/IEEE, 2023. doi:10.1109/ASE56229.2023.00030.

[13] Dan Mateas. The carbon cost of testing pipelines. https://www.green-coding.io/case-studies/carbon-cost-of-testing-pipelines/, 2024.

[14] Microsoft. Azure ip ranges and service tags – public cloud. https://www.microsoft.com/en-us/download/details.aspx?id=56519, 2025.

[15] Microsoft. Microsoft datacenters: Sustainability and efficiency. https://datacenters.microsoft.com/sustainability/efficiency/, 2026. Accessed: 2026-04-02.

[16] Nuno Saavedra, Alexandra Mendes, and João F. Ferreira. Environmental impact of CI/CD pipelines. arXiv preprint arXiv:2510.26413, 2025. URL: https://arxiv.org/abs/2510.26413.

[17] Todd Sedano, Paul Ralph, and Cécile Péraire. Software development waste. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 130–140, 2017. doi:10.1109/ICSE.2017.20.

[18] U.S. Department of Energy. Purchasing energy-efficient light bulbs, 2026. Accessed: 2026-04-02. URL: https://www.energy.gov/cmei/femp/purchasing-energy-efficient-light-bulbs.

[19] U.S. Environmental Protection Agency. eGRID 2023 summary tables, revision 2. Technical report, EPA, 2025. URL: https://www.epa.gov/system/files/documents/2025-06/summary_tables_rev2.pdf.

[20] U.S. Environmental Protection Agency. Greenhouse gas emissions from a typical passenger vehicle, 2025. Accessed: 2026-04-02. URL: https://www.epa.gov/greenvehicles/greenhouse-gas-emissions-typical-passenger-vehicle.

[21] U.S. Environmental Protection Agency. Greenhouse gas equivalencies calculator: Calculations and references, 2026. Accessed: 2026-04-02. URL: https://www.epa.gov/energy/greenhouse-gas-equivalencies-calculator-calculations-and-references.

[22] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, and Massimiliano Di Penta. Configuration smells in continuous delivery pipelines: A linter and a six-month study on GitLab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 327–337. ACM, 2020. doi:10.1145/3368089.3409709.

[23] Nimmi Rashinika Weeraddana, Mahmoud Alfadel, and Shane McIntosh. Dependency-induced waste in continuous integration: An empirical study of unused dependencies in the npm ecosystem. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. doi:10.1145/3660823.

[24] World Resources Institute and World Business Council for Sustainable Development. Ghg protocol scope 2 guidance: An amendment to the ghg protocol corporate standard, 2015. Accessed: 2026-04-02. URL: https://ghgprotocol.org/sites/default/files/standards/Scope%202%20Guidance_Final_Sept26.pdf.