GreenField - Cross Boundary JSON Field Analysis

Group 14

Preethika Ajaykumar, Atharva Dagaonkar, Riya Gupta, Sneha Prashanth, Deon Saji
Delft University of Technology
The Netherlands

1 INTRODUCTION

Modern web applications rely on JSON-based APIs for communication between frontend and backend components. In practice, backend services often return far more data than is required by the client, resulting in over-fetching. A study of open-source REST API clients found that, at the median, only around 6% of returned JSON fields were actually used, suggesting that switching to field-level queries could reduce transmitted data by up to 94% in field count and 99% in bytes [4]. This leads to redundant data transfer, increased bandwidth usage, additional serialisation overhead, and higher energy consumption.

Sustainable software engineering emphasises the importance of reducing unnecessary computation and data movement. Previous work has shown that data transfer and API design decisions have a measurable impact on the energy consumption of web applications [9, 13]. However, existing approaches primarily focus on coarse-grained optimisation, with limited support for identifying inefficiencies at the level of individual data fields exchanged between frontend and backend components.

To address this limitation, this paper introduces *GreenField*, a static analysis tool that detects unused JSON fields across frontend and backend codebases. The tool maps API endpoints, extracts response structures, and analyses field-level usage to identify data that is transmitted but never accessed. Integrated into Visual Studio Code, it provides actionable feedback during development and can be applied to real-world full-stack codebases.

We explore the following research questions:

> **RQ1:** To what extent do web applications transmit JSON fields that are not used?
>
> **RQ2:** How effectively can static analysis identify and reduce redundant JSON field transmission to improve efficiency and sustainability?

By addressing these questions, this work examines how fine-grained analysis of API data exchange can contribute to more efficient and sustainable application development.

2 BACKGROUND AND RELATED WORK

The growing importance of energy-aware software development provides the broader motivation for this work. Hindle [8] introduces the concept of green mining, linking software characteristics to energy consumption and advocating for tooling that supports sustainable development practices. Building on this, Verdecchia et al. [18] call for methodologies that treat energy efficiency as a first-class concern throughout the software lifecycle, while Calero and Piattini [5] offer a foundational account of what software sustainability means in practice. This framing situates GreenField within a wider movement toward making energy efficiency a genuine design priority in software engineering.

Within this context, research has examined energy consumption in web applications from the perspective of both infrastructure and application design. Pérez-Castillo and Piattini [13] present a systematic review identifying data transfer, computation, and resource utilisation as key contributors to energy usage, with inefficient data exchange shown to significantly increase the energy footprint of web systems. Similarly, Jagroep et al. [9] provide an empirical analysis of RESTful web services and demonstrate that API design decisions, particularly those affecting payload size and frequency of communication, have a measurable impact on energy consumption.

At the protocol level, this relationship has been further quantified by Saez et al. [17], who show that energy cost per byte varies directly with payload size, a finding that underscores the importance of minimising unnecessary data in each API response. Anwar et al. [2] reinforce this point in the context of mobile applications, demonstrating that varying message payload sizes produce measurable differences in HTTP energy consumption.

These studies collectively establish the link between software design and energy consumption, yet none explicitly address inefficiencies at the level of individual data fields in API responses. In REST-based architectures, fixed endpoints routinely return more data than a client actually needs [6], a problem known as over-fetching that results in redundant data transfer at scale. GreenField addresses this gap by enabling fine-grained analysis of JSON data exchanged between frontend and backend components. By identifying unused fields, it provides a practical mechanism for reducing redundant data transfer and improving the efficiency of web applications.

3 IMPACT ON SUSTAINABILITY

Unused JSON fields arise when backend services include data that is never accessed by the frontend. Although functionally harmless, these fields introduce inefficiencies at multiple stages of the data lifecycle. The data must still be serialised on the server, transmitted over the network, and parsed on the client, despite not contributing to application behaviour.

This redundant processing increases network traffic and bandwidth usage, which in turn leads to higher energy consumption. At scale, such inefficiencies accumulate and contribute to the environmental footprint of web systems. Figure 1 illustrates this causal chain, showing how seemingly small inefficiencies at the field level propagate into measurable sustainability impacts.

GreenField addresses these inefficiencies. The results are presented directly to developers, including estimates of wasted data

Figure 1: Impact of unused JSON fields: redundant data transfer increases network usage and energy consumption.

transfer. This enables targeted optimisation of API responses, reducing payload sizes and eliminating redundant data. As a result, network usage is lowered and energy consumption is reduced.

Figure 2 highlights how GreenField transforms previously hidden inefficiencies into actionable insights, allowing developers to improve both system performance and sustainability as part of the regular development workflow.

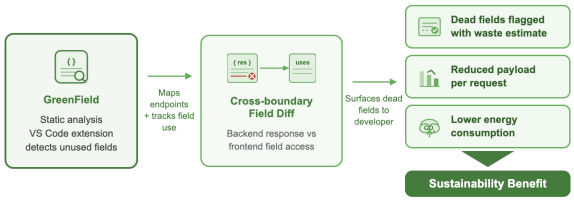

Figure 2: GreenField reduces redundant data transfer, lowering payload size and energy consumption.

Together, these figures highlight the transition from inefficiency to optimisation enabled by static analysis.

4 METHODOLOGY

GreenField operates across the application boundary, linking data production on the server side with data consumption on the client side. Fields that are transmitted but never accessed represent redundant data transfer that contributes to unnecessary network load and increased energy consumption. Prior research has demonstrated that payload size and frequency of communication are measurable drivers of the energy footprint of web services [9], and that the electricity intensity of fixed-line data transmission can be used to translate byte volume into energy estimates [3]. GreenField builds on these findings by making such inefficiencies visible within the programming environment, so that developers can address them during routine development rather than as a separate optimisation activity.

4.1 Static Analysis Pipeline

The implementation follows a multi-stage static analysis pipeline in which both frontend and backend source files are parsed into abstract syntax trees (ASTs) without executing the application. The pipeline proceeds through five stages: endpoint mapping, field extraction, usage tracking, identifying dead-fields, and sustainability scoring. Each stage operates on a distinct concern and communicates with adjacent stages through shared data structures defined in a common type interface (‘Endpoint‘, ‘Field‘, and ‘FieldSet‘). This separation allows backend language parsers to be developed and extended independently.

4.2 Cross-Boundary Endpoint Resolution

To determine whether a field is unused, the tool must connect the frontend API calls with the corresponding backend endpoints. GreenField identifies API calls in the frontend (e.g. `fetch` or `axios`) and matches them with route definitions in the backend.

Since different frameworks use different formats for defining routes, the tool normalises URL patterns into a common representation. This allows it to match endpoints across languages such as JavaScript, Python, Java, and Go. In cases where routes are defined across multiple files and cannot be fully reconstructed, the tool uses a fallback strategy described in Section 4.6.

4.3 Field Extraction

For each endpoint, GreenField extracts the fields that are sent by the backend and, where relevant, those included in frontend requests. The extraction process depends on the programming language used, but generally focuses on identifying JSON structures returned by backend routes or sent in API calls.

Regardless of the language, all fields are represented in a common format that includes the field name, its role (request or response), and its location in the code. This allows the tool to analyse fields consistently across different parts of the system.

4.4 Frontend Usage Analysis

The frontend analysis identifies which response fields are actually read by the client. The usage tracker records the names of fields accessed through the following patterns in TypeScript and JavaScript source files: direct property access, optional chaining, simple and aliased destructuring, nested destructuring, and template literals and JSX expressions, which resolve to property access expressions at the AST level.

A conservative policy is applied to dynamic bracket access (`obj[key]`); such expressions are excluded from the tracked set entirely. Any field that could plausibly be read through a computed key is therefore treated as potentially used and is not flagged as dead [15]. This avoids false positives at the cost of reduced recall in codebases that make heavy use of runtime key computation [3].

To reduce noise from non-API property accesses, such as calls to JavaScript built-in methods, DOM APIs, HTTP internals etc. a deny-list is applied. Names matching this list, names following an `ALL_CAPS` pattern, and single-character identifiers are excluded from the tracked set.

4.5 Dead-Field Detection and Waste Scoring

The diff engine computes dead fields as:

$$N_{\text{dead\_fields}} = \text{Defined}(\text{backend response}) - \text{Accessed}(\text{frontend}) \quad (1)$$

Fields present in the backend response payload but absent from the frontend tracked access set are classified as dead. Each dead field is assigned a waste score:

$$\text{waste\_score} = N_{\text{dead\_fields}} \times \hat{b} \times R \tag{2}$$

where $\hat{b}$ is an estimated byte size for the field's serialised value and R is the estimated daily request volume for the endpoint.

Field byte sizes are approximated using naming heuristics: timestamp fields are assigned 26 bytes, UUID identifiers 38 bytes, Boolean fields 5 bytes, token and hash fields 66 bytes, and general string fields 22 bytes. Daily request volume is estimated per endpoint pattern, with health-check endpoints assumed to receive 100,000 requests per day and administrative endpoints 500.

To estimate environmental impact, wasted data transfer is directly converted to carbon emissions using a data transfer carbon intensity factor:

$$E_{\text{day}} = \text{waste\_score} \times \gamma \tag{3}$$

where $\gamma = 6.9 \times 10^{-8}$ gCO_2e/byte represents the carbon intensity of data transfer excluding user devices.

This value is derived from the Sustainable Web Design Model (SWDM v4) by combining data center and network operational and embodied energy intensities (0.139 kWh/GB) and applying a global average carbon intensity of 494 gCO_2e/kWh, yielding approximately 69 gCO_2e/GB [1].

This formulation captures backend-attributable emissions from unnecessary data transfer. It intentionally excludes user device impacts to isolate the effects of server-side design decisions, making the metric suitable for comparative analysis and optimisation [1].

4.6 Global Fallback Analysis

For codebases in which the routing architecture prevents per-endpoint URL matching, most notably Go backend using gorilla/mux sub-routers, where path prefixes are defined across multiple files, GreenField falls back to a global analysis mode. In this mode, all backend response fields are compared against the union of all frontend-accessed field names, without path-level filtering. This produces an upper bound on the number of dead fields, some flagged fields may in fact be consumed by a different endpoint than the one that defines them.

4.7 Tool Integration

GreenField is integrated into Visual Studio Code as an extension, registering a `GreenField: Scan Workspace` command that triggers the full analysis pipeline. Results are surfaced through three primary components: (1) inline diagnostic warnings placed at the definition site of each dead field, reporting the field name and its estimated per-request waste; (2) a status bar item displaying the live count of mapped endpoints, dead fields found, and estimated wasted kilobytes per request; and (3) a dashboard presenting a per-endpoint breakdown of dead fields and their corresponding waste scores. A background scan mode re-runs the analysis on each file save, providing continuous feedback without interrupting the developer's workflow.

[1] Sustainable Web Design Model: https://sustainablewebdesign.org/estimating-digital-emissions/#faq

The sustainability rationale for this integration follows from prior work by Hindle [8], which argues that linking software characteristics to energy consumption requires tooling embedded in the development process rather than separate auditing steps.

4.8 Supported Stacks

Table 1 summarises the languages and frameworks currently supported by GreenField.

Table 1: Supported languages, frameworks, and HTTP client patterns.

Layer	Support
Frontend	TypeScript, JavaScript (React, Vue, Svelte, fetch/axios)
Backend	Node.js/Express, Python (Flask, FastAPI), Java (Spring Boot), Go (Gin, gorilla/mux, net/http)
HTTP clients	`fetch`, `axios`, `doFetch`-style wrappers
API style	REST / JSON

4.9 Architecture

Figure 3 illustrates the high-level architecture of GreenField, which is structurally divided into three interconnected layers:

- **Source Input:** Ingests the polyglot frontend and backend codebases.
- **Analysis Core:** Coordinates the language-specific AST parsers, cross-boundary endpoint resolution, and the diff and waste scoring engines.
- **Output Layer:** Routes the calculated sustainability metrics and dead-field diagnostics directly to the development workflow.

4.10 Standalone CLI Scanner

To support reproducible evaluation outside the VS Code environment, GreenField also provides a command-line scanner that executes the full analysis pipeline without requiring the extension host. The script accepts a target directory, collects all TypeScript, JavaScript, Python, Java, and Go source files, runs endpoint mapping, per-endpoint field-set analysis, and global fallback analysis, and prints a structured summary to standard output. It uses the same field extractors, usage trackers, diff engine, and waste-scoring heuristics as the extension, ensuring that results are numerically consistent between the two interfaces. The scanner was used to obtain the real-world evaluation results reported in Section 6.1.1.

5 EXPERIMENTS

We design two evaluations corresponding to each research question. The first uses a controlled benchmark to assess detection accuracy under known conditions. The second applies the tool to real-world open-source repositories to measure the prevalence of unused fields in practice [16].

5.1 Benchmark Evaluation for Synthetic Data

To evaluate how accurately GreenField identifies dead fields, we construct a suite of 20 synthetic full-stack projects, each with a known ground truth. The projects are generated programmatically

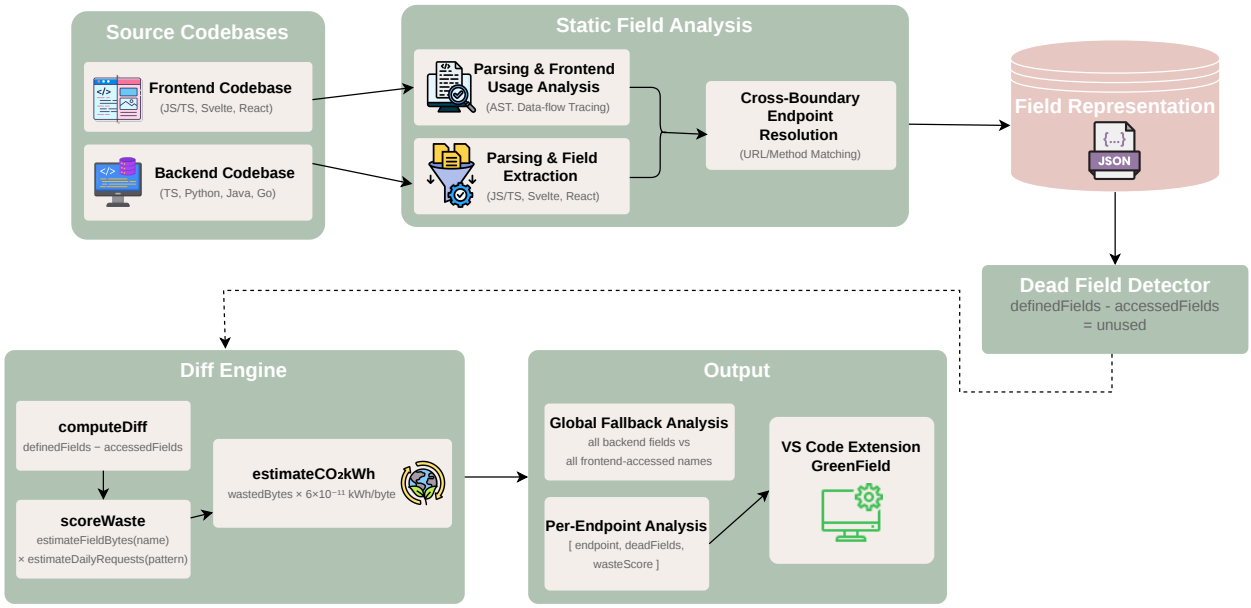

Figure 3: GreenField static analysis pipeline architecture. Source codebase inputs (frontend and backend) flow through endpoint mapping, field extraction, and usage analysis stages to produce dead field detection results with sustainability scoring.

from a compact specification using a generator script, which produces a backend file, a frontend file, and a JSON file containing the expected dead fields.

The 20 projects are distributed across three backend languages (TypeScript, Python, Java) and cover seven distinct frontend access patterns:

- **Direct property access** (e.g. `res.data.name`) - projects 01-03
- **Destructuring** (e.g. `const { name, email } = res.data`) - projects 04-06
- **Optional chaining** (e.g. `res.data?.name`) - projects 07-09
- **JSX field access** (e.g. `<p>{item.name}</p>`) - projects 10-11
- **Template literals** (e.g. `` `${data.firstName}` ``) - project 12
- **Dynamic bracket access** (e.g. `data[key]`) - projects 13-15
- **Python and Java backends** - projects 16-20

Projects 13 and 14 use purely dynamic field access; no fields should be flagged as dead because the static analyser cannot determine which keys are read at runtime [11]. Project 15 mixes one static access with further dynamic access; again, no fields should be flagged due to a conservative policy. For all remaining projects, the expected dead fields are explicitly specified.

Since GreenField's output is a binary classification, each field is either flagged as dead or not. We evaluate this using precision, recall, and F1-score [14]. Precision matters here because incorrectly flagging a live field as dead could lead a developer to remove data that is actually used [10]. Additionally, recall is important since an undetected dead field represents a missed opportunity for optimisation[19, 20]. The F1-score ties both together, which is useful given that the two types of metrics carry different practical consequences. We report both per-project and aggregate results.

5.2 Real-World Evaluation

To assess whether unused fields occur in practice and at what scale, we apply GreenField to two open-source applications representing different technology stacks.

freeCodeCamp [7] is an open-source learning platform with a TypeScript/React frontend and a Node.js/TypeScript backend. Its smaller API surface makes it well-suited for per-endpoint analysis, allowing a direct comparison of produced and consumed fields at the route level. The latest commit in our cloned repository is from 28th March 2026, 16:32. Running with updated commits can lead to different results.

Mattermost [12] is a large-scale team messaging platform with a React/TypeScript frontend and a Go backend. It uses gorilla/mux for routing with a two-level sub-router architecture that introduces additional complexity for endpoint resolution. The latest commit in our cloned repository is from 18th March 2026, 12:05. Running with updated commits can lead to different results.

6 RESULTS

6.1 RQ1: To what extent do web applications transmit JSON fields that are not used?

We answer RQ1 by analysing two production open-source codebases to assess whether dead-field transmission occurs in practice and, if so, at what scale.

6.1.1 Real-world scans. Table 2 summarises the scan results for the two production codebases.

Table 2: GreenField scan results for two production open-source codebases. Sustainability estimates use per-field byte heuristics and tiered daily request volumes (see Section 4.5).

Metric	Mattermost	freeCodeCamp
Files Scanned	6,758	1085
Endpoints mapped	457	7
Response fields	8,344	55
Dead fields	4,178	16
Dead-field rate	**50%**	**29%**
Wasted bytes/day	~1.22 GB	~3,520 KB
Estimated energy/day	73.1 Wh	0.21 Wh
Analysis method	Global fallback	Global fallback

Global fallback analysis was used for both codebases. For Mattermost, this is because its two-level gorilla/mux sub-router architecture distributes path segments across multiple files, preventing reliable per-endpoint URL reconstruction; the 50% rate is therefore an upper bound. For freeCodeCamp, per-endpoint matching produced zero linked endpoint pairs, so the global fallback was also applied; 16 extracted backend response fields were absent from the frontend-accessed name set, yielding a 29% dead-field rate.

Dead-field composition. Figure 4 shows the 20 most frequently occurring dead field names across Mattermost's Go backend files. Generic structural fields (`type`, `value`) dominate, but internal and administrative fields such as `logger` and `rootStore` also appear prominently, suggesting that server-side implementation details can leak into API responses as codebases evolve.

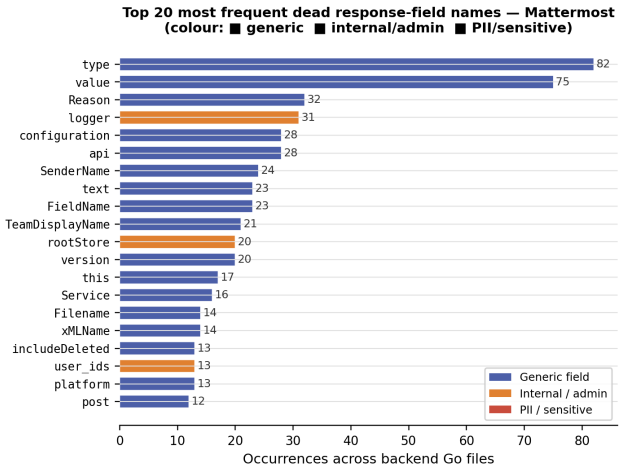

Figure 4: Top 20 dead response-field names by occurrence frequency in Mattermost's Go backend.

> **Answer to RQ1:** In the two full-stack codebases studied, more than 25% response fields were transmitted but never consumed by the frontend. This represents avoidable bandwidth and energy overhead that scales linearly with request volume.

6.2 RQ2: How effectively can static analysis identify and help reduce redundant JSON field transmission to improve efficiency and sustainability?

Whereas RQ1 establishes that dead-field transmission occurs in practice, RQ2 evaluates whether static analysis is an effective way to detect and prioritise this waste for remediation.

6.2.1 Detection accuracy on the synthetic benchmark. To assess whether static analysis can reliably identify dead JSON fields, we evaluated GreenField on the synthetic benchmark of 20-projects described in Section 5.1.

GreenField achieved 100% precision and recall across all 20 projects with no false positives.

6.2.2 Cross-language coverage. Figure 5 illustrates the semantic distribution of the 60 detected dead fields across four categories: internal and administrative fields (18, 30%), metadata (16, 27%), business data (16, 27%), and Personally Identifiable Information (PII) such as email addresses and phone numbers (10, 17%). The presence of PII in this distribution is particularly notable from a sustainability perspective: unnecessary transmission of personal data increases both payload size and exposure risk without contributing to frontend functionality.

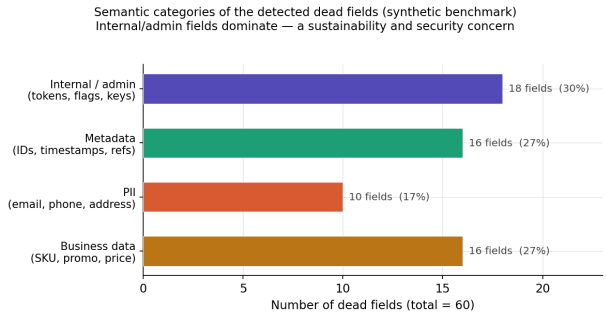

Figure 5: Semantic distribution of the 60 dead fields detected in the synthetic benchmark.

For **Python**, the regex-based extractor correctly recovers Pydantic field names from `BaseModel` definitions (benchmark project p-19: all three dead fields detected with zero FP/FN). For **Java**, the extractor correctly resolves annotation-based field renaming by using JsonProperty, mapping the Java identifier to its serialised name rather than the source variable name (p-20: all four dead fields detected). This cross-language reliability is a prerequisite for applying the tool to the polyglot stacks typical of production services.

6.2.3 Sustainability impact quantification. Beyond identifying dead fields, GreenField quantifies their estimated sustainability impact using the waste-scoring model from Section 4.5, surfacing concrete numbers to motivate remediation and enable developers to prioritise the highest-impact fields. Figure 6 shows how the

real-world estimates project across request volumes. The model scales linearly with traffic, so even modest per-request inefficiencies become material at high traffic. All estimates rely on the energy coefficient from Aslan et al. [3] and are model-based projections, not direct measurements.

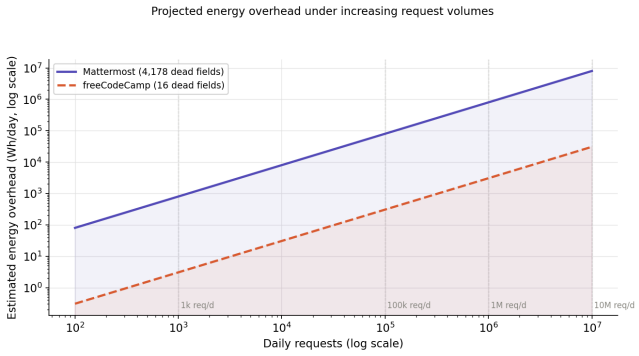

Figure 6: Projected energy overhead at increasing request volumes for both real-world codebases (log–log axes). Vertical dashed lines mark representative traffic levels from 1k to 10M requests/day.

Prioritisation: Pareto structure of waste. Not all dead fields contribute equally to bandwidth overhead. Figure 7 shows the cumulative distribution of waste scores for Mattermost's 4,178 dead fields, sorted from highest to lowest. The distribution is notably uneven: removing the top 30% of fields by waste score eliminates 50% of total waste, while the top 69% accounts for 80%. This Pareto structure means that a targeted remediation effort focusing on the highest-score fields can recover the majority of wasted bandwidth with a fraction of the overall cleanup effort, making the tool's output directly actionable for sustainability-driven development.

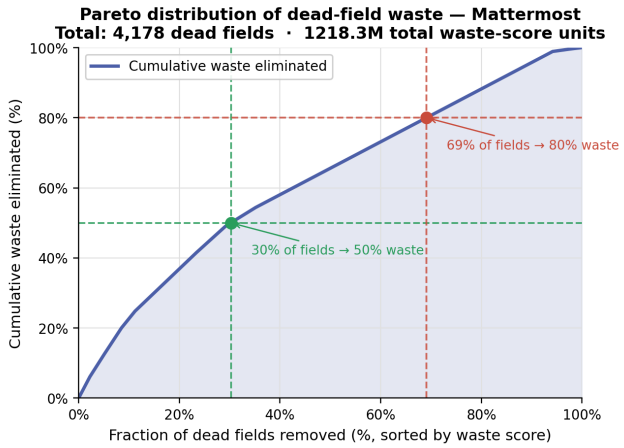

Figure 7: Pareto distribution of dead-field waste scores in Mattermost.

Answer to RQ2: GreenField achieves 100% precision and recall on the synthetic benchmark across all supported stacks and seven frontend access patterns, confirming that static cross-boundary analysis reliably identifies this class of inefficiency. Applied to production codebases, the tool converts dead-field counts into concrete sustainability estimates and surfaces them as inline editor feedback. Precision is bounded in practice by conservative handling of dynamic property access and a global-fallback mode required for codebases with distributed routing architectures.

7 DISCUSSION

The results indicate that dead JSON field transmission is a recurring sustainability inefficiency embedded in everyday web development, rather than a code-quality concern at the margins. Across the two production systems analysed, a substantial share of transmitted response fields were never consumed by the frontend, suggesting that over-fetching is a systematic pattern rather than an isolated anomaly.

7.1 Environmental sustainability: avoidable digital waste

Each unnecessary field must still be serialised on the server, transmitted over the network, and parsed on the client, despite contributing nothing to user-visible functionality. The cost of a single unused field is negligible, but this waste accumulates meaningfully under production traffic and grows linearly with request volume, making even modest inefficiencies relevant at scale.

This aligns with a core idea from Sustainable Software Engineering: sustainability problems often arise not from dramatic failures, but from repeated low-level inefficiencies that remain invisible during normal development. GreenField makes one such inefficiency visible and quantifiable.

7.2 Social impact: responsible data minimisation

The semantic breakdown of dead fields in the synthetic benchmark shows that the waste is not restricted to harmless structural padding. Internal and administrative fields appear frequently alongside personally identifiable information such as email addresses and phone numbers. Transmitting such fields without frontend use undermines data minimisation and broadens the exposure surface of the system without any functional benefit, making this a concern that extends beyond performance alone.

The accessibility of the tool also contributes to its social dimension. GreenField [2] is open source and distributed as an installable VS Code extension, meaning any development team can adopt it without cost or infrastructure changes. This lowers the barrier to privacy-conscious development, making responsible data minimisation a realistic practice for small teams and individual developers, not only organisations with dedicated security resources.

7.3 Lifecycle perspective: how waste persists

The freeCodeCamp result is particularly instructive. The dead fields identified there originated from test plugin fixtures compiled into the API bundle, illustrating how development scaffolding can leak

[2] https://github.com/riyagupta0701/GreenField/

into production responses. Waste can be introduced through shortcuts, defaults, generated artefacts, and legacy compatibility decisions, then persist unnoticed unless tooling exposes it, which points to a broader principle that sustainability must be considered across the full software lifecycle, not only at deployment time.

The Mattermost results show the same dynamic at larger scale. Payload bloat in a mature codebase is less a one-off implementation mistake than a maintenance phenomenon that accumulates as systems evolve.

7.4 Actionability and prioritisation

The Pareto analysis reveals that waste is unevenly distributed, which has a useful practical implication: teams do not need to eliminate every dead field to make a meaningful dent. Targeting the highest-impact fields first recovers a substantial share of wasted bandwidth with comparatively limited effort, making GreenField's output directly actionable rather than merely diagnostic.

7.5 Implications for sustainable software engineering

GreenField translates an otherwise hidden class of inefficiency into something measurable and addressable inside the development workflow. By doing so, it touches all three sustainability dimensions: environmental, through reduced computation and data transfer; social, through better data minimisation practices; and individual, through reduced interface complexity for developers. Dead-field detection is a concrete example of how sustainability considerations can be woven into regular engineering work rather than deferred to a separate audit.

8 LIMITATIONS

GreenField is constrained by limitations inherent to static analysis. Dynamic property access prevents precise resolution of field usage at runtime. To avoid incorrect warnings, the tool adopts a conservative strategy, treating such fields as potentially used. This reduces false positives, but it can lead to missed optimisation opportunities in codebases that rely heavily on dynamic patterns. This limitation could be mitigated through hybrid approaches that incorporate lightweight runtime instrumentation to capture actual field usage.

Accurate endpoint matching is also challenging in systems with distributed routing logic. In frameworks where routes are composed across multiple files, precise reconstruction of endpoint mappings is not always possible. In such cases, GreenField falls back to a global comparison between backend fields and frontend usage, meaning that reported dead-field counts should be interpreted as approximate rather than exact. Improved static analysis of routing frameworks or build-time route resolution could reduce reliance on this fallback.

The sustainability estimates are derived from a simplified model based on assumed field sizes, request volumes, and a standard energy coefficient for data transfer [3]. These values provide an indication of relative impact, but they are not direct measurements of runtime energy consumption. Incorporating empirical measurements would strengthen the accuracy of these estimates.

Additionally, the study does not evaluate how developers respond to the tool in practice, leaving open the question of its long-term impact on development behaviour. Furthermore, the evaluation is limited to two open-source systems. Although these differ in scale and architecture, a broader set of applications would provide stronger evidence of generalisability.

Nevertheless, the consistency of results across both systems, combined with perfect accuracy on controlled benchmarks, suggests that the observed patterns of redundant data transfer are not incidental but indicative of a broader phenomenon.

9 CONCLUSION

This paper presented GreenField, a static analysis tool for detecting unused JSON fields across frontend and backend codebases. Such fields represent a subtle but recurring form of inefficiency, as they increase payload size, consume network and processing resources, and may expose unnecessary data without contributing to application behaviour.

The evaluation shows that this issue is both detectable and prevalent. GreenField achieved perfect accuracy on the synthetic benchmark and identified substantial levels of unused fields in real-world systems, with more than 25% of response fields not being consumed. These findings indicate that over-fetching is not incidental, but a systematic pattern that can contribute to avoidable resource usage.

From a sustainability perspective, reducing redundant data transfer reduces network utilisation and associated energy consumption. Although the estimates presented are model-based, they illustrate how small inefficiencies at the field level can scale into measurable impact under realistic workloads. By making such inefficiencies visible during development, GreenField supports more resource-aware design decisions.

This work demonstrates that fine-grained analysis of API data exchange can serve as a practical mechanism for improving both efficiency and sustainability in web applications, without requiring changes to the underlying infrastructure.

9.1 Future Work

Future work should focus on strengthening both the analysis and its empirical grounding. Incorporating runtime measurements would improve the accuracy of sustainability estimates and validate the model against real system behaviour. The analysis could also be extended to better handle dynamic access patterns, for example, through hybrid approaches that combine static and runtime information.

Expanding support beyond REST-based JSON APIs to other paradigms such as GraphQL and gRPC would improve applicability across modern systems. Additionally, evaluating the tool through repeated measures over an extended period would help determine whether developer-facing feedback leads to sustained reductions in redundant data transfer in practice.

REFERENCES

[1] Anders Andrae. 2020. New perspectives on internet electricity use in 2030. *Engineering and Applied Science Letters* 3 (06 2020), 19–31. https://doi.org/10.30538/psrp-easl2020.0038

[2] Hina Anwar, Dietmar Pfahl, and Satish Narayana Srirama. 2018. An Investigation into the Energy Consumption of HTTP POST Request Methods for Android App Development. In *Proceedings of the 13th International Conference on*

Software Technologies (ICSOFT). SCITEPRESS, 241–248. https://doi.org/10.5220/0006846102410248

[3] 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

[4] Gleison Brito and Marco Tulio Valente. 2020. Migrating to GraphQL: A Practical Assessment. In *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 140–150. https://doi.org/10.1109/SANER48275.2020.9054835

[5] Coral Calero and Mario Piattini. 2017. Puzzling out Software Sustainability. *Sustainable Computing: Informatics and Systems* 16 (2017), 117–124. https://doi.org/10.1016/j.suscom.2017.10.011

[6] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-Based Software Architectures*. Ph.D. Dissertation. University of California, Irvine. https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[7] freeCodeCamp. 2026. freeCodeCamp. https://github.com/freeCodeCamp/freeCodeCamp. (2026). GitHub repository, accessed 25 March 2026.

[8] Abram Hindle. 2012. Green Mining: A Methodology of Relating Software Change and Configuration to Power Consumption. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 78–87. https://doi.org/10.1109/MSR.2012.6224300

[9] Erik A. Jagroep, Jan Martijn E. M. van der Werf, Sjaak Brinkkemper, Giuseppe Procaccianti, Patricia Lago, Leen Blom, and Rob van Vliet. 2016. Software Energy Profiling: Comparing Releases of a Software Product. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE Companion)*. ACM, 523–532. https://doi.org/10.1145/2889160.2889219

[10] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE, 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[11] Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM, 271–280. https://doi.org/10.1145/1081706.1081759

[12] Mattermost. 2026. Mattermost. https://github.com/mattermost/mattermost. (2026). GitHub repository, accessed 25 March 2026.

[13] Ricardo Pérez-Castillo and Mario Piattini. 2014. Analyzing the Harmful Effect of God Class Refactoring on Power Consumption. *IEEE Software* 31, 3 (2014), 48–54. https://doi.org/10.1109/MS.2013.163

[14] David M. W. Powers. 2011. Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness and Correlation. *Journal of Machine Learning Technologies* 2, 1 (2011), 37–63.

[15] Jerin Anan Proma, Fatema Zannat, Mohammed Aftab, Tahura Tripty, M. Khan, Raihan Islam, and Rashedul Amin Tuhin. 2025. Energy-Efficient Web Design: Measuring Impact of Front-end Optimizations. 92–100. https://doi.org/10.1145/3777555.3777561

[16] Dag I. K. Sjøberg, Tore Dybå, and Magne Jørgensen. 2005. A Survey of Controlled Experiments in Software Engineering. In *Proceedings of the 2005 International Symposium on Empirical Software Engineering*. IEEE, 59–68. https://doi.org/10.1109/ISESE.2005.1541819

[17] Sergio Sáez, Martino Trevisan, Idilio Drago, and Daniela Giordano. 2023. Energy Consumption of Smartphones and IoT Devices When Using Different Versions of the HTTP Protocol. *Pervasive and Mobile Computing* 96 (2023), 101871. https://doi.org/10.1016/j.pmcj.2023.101871

[18] Roberto Verdecchia, Patricia Lago, Christof Ebert, and Carol de Vries. 2021. Green IT and Green Software. *IEEE Software* 38, 6 (2021), 7–15. https://doi.org/10.1109/MS.2021.3102254

[19] N. Wasif. 2024. Green Software Engineering: A Pathway to Sustainability in Renewable Energy Systems. *ResearchGate (preprint)* (2024). Available online: https://www.researchgate.net/publication/384660032.

[20] W. Wysocki, I. Miciuła, and P. Plecka. 2025. Methods of Improving Software Energy Efficiency: A Systematic Literature Review and the Current State of Applied Methods in Practice. *Electronics* 14, 7 (2025), 1331. https://doi.org/10.3390/electronics14071331

A DECLARATIVE USE OF GENERATIVE AI

Large language models (ChatGPT and Claude) were used to rephrase text and improve style, structure, and grammar. They were not used to generate new content, but rather to improve the overall clarity, consistency, and readability of the report. In addition to LLMs, Grammarly has been used to correct any grammar mistakes.

Examples of prompts used:

- (ChatGPT) "How can this sentence be rephrased to sound more stylistically correct?"
- (ChatGPT) "Please correct the grammar and improve the flow of this paragraph."
- (Claude) "Make this explanation more concise without changing its meaning."