AI/ML Driven Software Optimisation
to Reduce Cost and Climate Impact

DELIVERABLE D3.1

(2025) Annual SotA review of the static code analysis and QA tools, with focus on NLP/GenAI solutions and the application of green considerations as a quality metric

— Restricted —

Authors:

Andreas Jedlitschka (WP3 Leader), Michele Albano - Aalborg Universitet, Chris Dean - Digital Tactics, Asger Posborg Jæhger - Edora A/S, Adam Trendowicz - Fraunhofer, Anna Maria Vollmer - Fraunhofer, Julien Siebert - Fraunhofer, Patricia Kelbert - Fraunhofer, Sven Theobald - Fraunhofer, Violeta Bonet Vila, Fraunhofer, Daniel Carlos Do Vale Ramos - ISEP, Pedro Faria - ISEP, Rita Inês Correia Da Costa - ISEP, Zita Vale - ISEP, Amir Soltani - KAN Engineering, Alireza Mehri - KAN Engineering, Gunel Jahangirova - Kings College, Jie Zhang - Kings College, Mohammad Mousavi - Kings College, Hugo Araujo - Kings College, Marcos Checa - Panel Sistemas, Cosme Gonzalez - Panel Sistemas, Christian Koerner - Siemens Germany, Ezgi Sarikayak - Siemens Germany, Matthias Saft - Siemens Germany, Angel Cataron - Siemens Romania, Fridtjof Siebert - Tokiwa, Michael Lill - Tokiwa, Daniel Esteban Villamil - UC3M, Juanmiguel Gomez - UC3M, Andrea Pabon - UC3M, Haluk Gokmen - VBT, Osman Çaylı - VBT, Sinan Kılıç - VBT, Yücel Şentürk - VBT, Constantin Deneke - ZAL Aero, Johannes Passand - ZAL Aero, Mario Paja - ZAL Aero, Steffen Ruesch - ZAL Aero, Stephan Rediske - ZAL Aero

	ITEA Call 2023
Project	23016 GreenCode
Version, Date	V1.0 23.12.2025

Version history

Version	Authors	Content	Date
0.1	Andreas Jedlitschka	Document creation and document structure	01.06.2025
0.2-0.7	Diverse authors	Individual contributions	31.10.2025
0.75	Patricia Kelbert	Reorganization of document structure	10.10.2025
0.8	All authors	Initial version ready for internal review	30.11.2025
0.9	All authors	Final review of individual contributions	16.12.2025
1.0	Andreas Jedlitschka and Daniel Villamil	Review, cosmetics, and finalization	23.12.2025

Content

Figures

Tables

1 Introduction

The GreenCode project is focused on AI/ML-driven software optimisation to reduce the cost and climate impact of software systems (software and the infrastructure it runs upon), and by implication the carbon impact of the IT sector at scale.

This is a large and growing subject area with various actors also now taking steps to improve elements of software sustainability and the efficiency of the software development lifecycle (SDLC) through education and AI interventions, some notable names being: the Green Software Foundation[1] (GSF) who focusses on international standards, best practice policy and community building; academic organisations such as the Software Sustainability Institute (SSI) at The University of Edinburgh[2] and the Digital Sustainability Centre at Vrije University, Amsterdam[3] who focus on education and outreach; numerous startups and other RD&I initiatives like the GENIUS[4] project focussing on AI tools for the SDLC.

Instead of replicating the efforts already undertaken by other organisations, we aim to build upon their work alongside our own initiatives to address the interconnected topics of software quality, sustainability, and performance improvement. Our goal is to offer a clear, measurable pathway towards reducing the energy consumption of software systems and certifying them according to established international standards, such as the Software Carbon Intensity (SCI) benchmark. Specifically, we are developing a modular AI application pipeline, accompanied by supporting tools, to automate the optimisation of software systems for both quality and energy efficiency. This approach allows for continuous tracking and upgrading in line with advances in the field, ensuring our solutions remain aligned with the current state of the art.

Furthermore, given that 60-80% of all software is regarded as legacy software and given that many market players are focussed on the generation of new software applications from scratch through AI, we intend to address the low-hanging fruit of legacy system maintenance, rationalisation, optimisation and upgrade/porting.

Our work aims to deliver measurable reductions in total cost of ownership (TCO) and technical debt for owners of existing systems, as well as verified green credentials and other benefits that may enhance market competitiveness. From a climate perspective, software optimisation can contribute to reducing emissions and energy consumption efficiently, particularly when applied to applications already deployed at scale.

This document reviews the latest in static code analysis and QA tools, emphasising NLP and GenAI solutions, and includes green considerations as a quality metric.

[1] https://greensoftware.foundation/
[2] https://www.software.ac.uk/
[3] https://vu.nl/en/about-vu/research-institutes/amsterdam-sustainability-institute/more-about/digital-sustainability-center
[4] https://itea4.org/project/genius.html

2 Thematic Review

The following sections present the current state of the art across the project's core themes, as identified through a comprehensive review of business and technology domains. The document complements deliverable “D2.1 State-of-the-Art" with specific themes, which form the basis for gap analysis and later for the specification of innovative business models:

- Task 3.1: Mapping of software system's codebase
- Task 3.2: Sustainability/energy focussed static code analysis and report
  - Static Code Analysis with focus on Energy
  - Benchmark Analysis
- Task 3.3: Code Artefact Generation
  - Test Code Generation
  - Documentation Generation
- Task 3.4: Quality Assessment of GenAI outcomes
- Task 3.5 Analysis of code artefacts as quality metrics
- Task 3.6: Static code analysers for compiled software

2.1 Mapping of software system's codebase

Modern software systems are vast and complex, often spanning millions of lines of code developed by large teams over many years. Documentation is frequently incomplete or outdated, making the source code itself the only reliable reference for understanding system behavior. As a result, “codebase mapping” – the creation of high-level, structured representations from raw source code – has become essential for effective maintenance, modernization, and quality assurance.

Key Findings

- Visual and Structural Mapping Techniques: A significant body of research focuses on creating high-level representations of software to aid human understanding, often employing metaphors to make complex systems more intuitive. The CodeSurveyor tool is a prime example, generating an interactive map where architectural components appear as continents and source files as countries. This method's sophistication lies in its underlying technique – a composition of force-directed graph layout and Voronoi tree-mapping – which demonstrates proven scalability by mapping massive codebases like the Linux kernel (1.4 MLOC) in just 1.5 minutes[5]. These approaches help developers quickly comprehend complex architectures and have demonstrated scalability on projects as large as the Linux kernel.
- Architecture Recovery via Dependency Analysis: Techniques that analyze dependencies between code elements can group related components into modules[6], providing a high-level architectural view. Integrating structural, semantic, and directory information leads to more accurate clustering and a better understanding of system structure.
- Systematic Reviews of Parallel Code Analysis Domains: Several mature research areas complement codebase mapping:

[5] N. Hawes, S. Marshall, C. Anslow (2015) "*CodeSurveyor: Mapping large-scale software to aid in code comprehension*," 2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT), Bremen, Germany, pp. 96-105, doi: 10.1109/VISSOFT.2015.7332419.

[6] S.P.R. Puchala, J.K. Chhabra, A. Rathee (2022) "*Software Architecture Recovery Using Integrated Dependencies Based on Structural, Semantic, and Directory Information*", International Journal of Information System Modeling and Design, Volume 13, Issue 1, ISSN 1947-8186, https://doi.org/10.4018/IJISMD.297060.

- Software Test-Code Engineering: A systematic mapping by Garousi et al.[7] classified 60 studies to provide a comprehensive overview of trends and techniques in test-code development and quality assessment;
- Metrics-Based Clone Detection: A review by Rattan & Kaur[8] analysed techniques that use software metrics to identify similar or identical code fragments, which are critical to manage for system maintenance and quality;
- Log and Source Code Matching: A study by Bushong et al.[9] systematically reviewed methods for matching information from program logs and stack traces back to the source code, a crucial step for fault localization;
- Code Clone Management: A systematic literature review by Kaur et al.[10] identified tools and methods for managing code clones through refactoring, a key activity for improving overall code quality;

➢ AI for Code Understanding: Systematic mapping studies also exist for the application of artificial intelligence to source code understanding tasks, indicating a mature and active area of research[11].

While these domains offer powerful tools, each address only part of the overall challenge. For example, visual tools are optimized for human understanding, while architectural recovery lacks the precision needed for automated refactoring.

2.1.1 Gaps Identified

➢ Existing approaches do not provide a universal, machine-readable model of code structure suitable for automated optimization and transformation.
➢ There is a lack of integration between high-level visualizations and the detailed, formal representations required for automated tools.
➢ Current methods often require manual intervention or are not scalable to the largest, most complex codebases.

2.1.2 Summary and Future Opportunities for GreenCode

The GreenCode project's implementation of a model-based reverse engineering approach, powered by ANTLR4 (more details in the annex), provides a strong and reliable framework for automated, precise, and standardised mapping of codebases spanning several programming languages. This strategy not only simplifies the process of extracting formal models from source

[7] V. Garousi, Y. Amannejad, A.B. Can (2015) "*Software test-code engineering: A systematic mapping*", Information and Software Technology, Volume 58, Pages 123-147, ISSN 0950-5849, https://doi.org/10.1016/j.infsof.2014.06.009.
[8] D. Rattan, J. Kaur (2016) "*Systematic Mapping Study of Metrics based Clone Detection Techniques*", AICTC '16: Proceedings of the International Conference on Advances in Information Communication Technology & Computing, Bikaner, India, https://doi.org/10.1145/2979779.2979855.
[9] V. Bushong, R. Sanders, J. Curtis, M. Du, T. Černý, K. Frajták, M. Bures, P. Tisnovsky, D. Shin, Dongwan (2020) "*On Matching Log Analysis to Source Code: A Systematic Mapping Study*". RACS '20: Proceedings of the International Conference on Research in Adaptive and Convergent Systems, Gwangju, Republic of Korea, pp. 181-187. https://doi.org/10.1145/3400286.3418262.
[10] M. Kaur, D. Rattan, M. Lal (2025) "*Insight into code clone management through refactoring: a systematic literature review*", Computer Science Review, Volume 58, 100767, ISSN 1574-0137, https://doi.org/10.1016/j.cosrev.2025.100767.
[11] D.R. Fudholi, A. Capiluppi (2025) "*Artificial intelligence for source code understanding tasks: A systematic mapping study*", Information and Software Technology, Volume 189, 107915, ISSN 0950-5849, https://doi.org/10.1016/j.infsof.2025.107915.

code but also guarantees high accuracy and seamless interoperability, underpinning the entire GreenCode workflow from quality assurance through to AI-driven optimisation.

Cobol language is widely used in IBM mainframe systems but it's not among the languages fully supported by ANTLR4. Even if there are Cobol85 grammar available for use with ANTLR4, because IBM mainframe Cobol includes specific structures such as z/OS extensions, compiler directives, conditional compilation, EXEC CICS/SQL constructs, GreenCode will include a Preprocessing & Normalization step for IBM mainframe Cobol sources, to process these structures before ANTLR parsing.

Looking to the future, there is significant potential to enhance SACA (Static Analysis of Code with ANTLR 4) by broadening its support to encompass additional programming languages, adopting more detailed feature extraction for in-depth analysis, and harnessing these comprehensive models to facilitate sophisticated predictions and improvements in energy consumption much earlier in the software development process.

2.2 Static Code Analysis with focus on Energy

2.2.1 Background

Traditional work on estimating a program's energy consumption (EC) either measures it with a profiler or predicts it dynamically. Both approaches require running the software – often with representative inputs, sufficient time, and an available platform. These methods yield accurate results but are computationally expensive, time-consuming and applicable in late phases of software development process, when software can be run. Dynamic prediction typically returns a single total-energy value; few approaches provide fine-grained estimates or EC time series. One solution to this problem is predicting software runtime energy consumption based on the analysis of software artifacts, such as source code, available already in the early stages of software development process.

Historically, these approaches seldom targeted Python, which was less popular when they were developed. In recent years, however, Python has become the most widely used programming language. According to the PopularitY of Programming Languages (PYPL) index from September 2025[12], Python held a 29.69% share compared with 14.72% for Java and 9.27% for C/C++. AI projects predominantly use Python thanks to its rich libraries, ease of use and interpretation, and cross-platform support, among other advantages[13]. The AI Index Report 2025 notes that AI-related GitHub repositories grew from 1,549 in 2011 to approximately 4.3 million in 2024[14], underscoring Python's importance as a target.

2.2.2 Current State of the Art

A systematic literature review[15] was conducted to explore existing methods for predicting software energy consumption through static code analysis. The review was structured around six research questions, covering software types, features used for prediction, model outputs, predictive models, measurement tools, and operational environments. The Scopus database was used as the primary source, complemented by backward and forward snowballing from seed papers. Publications were identified through a carefully constructed search query, screened for relevance, and assessed

[12] Popularity of Programming Language (PYPL): https://pypl.github.io/PYPL.html. [Accessed: 2025-10-15].
[13] Michael Iyam. The Importance of Python in Artificial Intelligence. https://michael-lyamm.medium.com/the-importance-of-python-in-artificial-intelligence-341c7af1fb94. [Accessed: 2025-12-11].
[14] Nestor Maslej et al. Artificial Intelligence Index Report 2025. 2025. arXiv: 2504.07139 [cs.AI]. url: https://arxiv.org/abs/2504.07139.
[15] To be published.

against predefined criteria. Full texts of relevant studies were reviewed to extract data addressing the research questions. Details with citations are provided in the annex.

2.2.3 Key Findings

- Diversity of Software and Languages Studied

Most studies concentrate on benchmark suites such as AnghaBench[16], BEEBS[17], PolyBench[18], Rodinia[19], and SPEC CPU2006[20] (retired in Jan. 2018), while also considering GPU-intensive workloads and Android applications. C and C++ are the primary languages examined, especially in the context of desktop, server, and GPU-based programs. Java is frequently used for analyses related to mobile environments, whereas Python and Fortran are less commonly featured. In some cases, the programming language is not explicitly stated, particularly in research where user behaviour is the central aspect being investigated.

- Features Used for Energy Prediction: There are three primary categories of input features:
    - Utilization-based: Measures of hardware resource consumption, such as CPU, memory, and cache usage;
    - Event-based: Information derived from system activities like system calls, protocol transitions, and performance counters; event-based features are used most frequently, followed by utilization-based and then code-analysis-based features. The granularity of prediction differs: most models estimate energy use at the application level, while a smaller number focus on kernels, basic blocks, or specific time intervals.
    - Code-analysis-based: Static attributes extracted from code, including opcode counts, LLVM characteristics, and software metrics.
- Model Outputs: Most models predict power (watts) or energy (joules), while some estimate performance metrics like execution time or instructions per second. A few studies focus on worst-case metrics (WCET, WCEC), particularly for embedded or real-time systems.
- Types of Predictive Models: There are five main model categories: linear, tree-based, neural networks, kernel-based, and distance-based. Linear models offer simplicity and interpretability, while advanced options like ensembles and neural networks address complex patterns. Hybrid models improve accuracy and clarity. Event-based models are used most often, followed by code-analysis and utilization-based types.
- Static vs. Dynamic Prediction: Most studies rely on dynamic prediction (via software execution), while fewer use static analysis (code inspection without running). Some models handle both static and dynamic inputs.
- Measurement Methods: Although software-based energy measurement tools like perf, NVML, and Intel Power Gadget are widely favoured for their flexibility and convenience, they tend to be less precise than hardware-based approaches. In contrast, hardware-based measurements – which rely on PMUs or external sensors – provide greater accuracy but

[16] A. Faustino da Silva et al. „ANGHABENCH: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction". In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2021, pp. 378‐390. doi: 10.1109/CGO51591.2021.9370322.

[17] J. Pallister, S. Hollis, and J. Bennett. BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms. 2013. arXiv: 1308.5174 [cs.PF]. url: https://arxiv.org/abs/1308.5174.

[18] Pouchet Louis-Noel. Polybench: The polyhedral benchmark suite. https://www.cs.colostate.edu /~pouchet/software/polybench/. [Accessed: 2025-10-13]. 2012

[19] Shuai Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: 2009 IEEE International Symposium on Workload Characterization (IISWC). 2009, pp. 44‐54. doi: 10.1109/IISWC.2009.5306797.

[20] Standard Performance Evaluation Corporation. SPEC CPU 2006 benchmark (Retired: January 2018). https://www.spec.org/cpu2006/. [Accessed: 2025-10-13].

are used less frequently due to their complexity and higher cost. Additionally, some research utilises hybrid or simulation-based tools.

- Evaluation Platforms: Research on energy consumption spans many types of platforms, including microcontrollers, Android smartphones, embedded ARM clusters, GPUs, servers, and consumer electronics. Among these, microcontrollers and mobile devices are studied most often, highlighting how crucial energy efficiency is in those areas.

2.2.4 Gaps Identified

- Limited focus on static code analysis: Most existing models rely on dynamic features (i.e., data collected during program execution), with relatively few studies exploring purely static, code-based prediction methods.
- Lack of standardization: There is no unified approach for feature selection, model evaluation, or reporting, making it difficult to compare results across studies.
- Underrepresentation of certain languages and platforms: Python, Fortran, and some embedded platforms are rarely studied, despite their growing importance – especially Python in data science and AI.
- Accuracy vs. practicality trade-off: Hardware-based measurement methods are more accurate but less practical for widespread use; software-based methods are more common but generally less precise.
- Granularity limitations: Most models predict energy consumption at the application level. Finer-grained predictions (e.g., at the function or basic block level) are much less common, limiting the ability to pinpoint energy-intensive code segments.
- Hybrid and multimodal approaches are rare: Few studies combine static and dynamic features or support both types of prediction within a unified framework.
- Limited transparency in model specification: Some studies do not provide detailed descriptions of their models, especially hybrid or multimodal approaches, making reproducibility and evaluation challenging.
- Environmental diversity: While a range of platforms is covered, certain environments – such as real-time embedded systems and heterogeneous clusters – are less frequently addressed.

2.2.5 Summary and Future Opportunities for GreenCode

Analysis of the related literature indicates several gaps, which create opportunities for research and development in the GreenCode project. The main opportunity concerns estimating and optimizing the energy consumption of data-intensive software applications. This requires addressing the deficits regarding the fine-grained static analysis of Python source code and creating explainable energy consumption prediction models.

GreenCode presents several opportunities for advancing energy-efficient software development. It can conduct static analysis of Python code, systematically benchmark energy consumption across extensive sets of real and LLM-generated code samples to establish robust baseline measurements, enable fine-grained predictions of energy usage, and support the development of transparent, explainable prediction models tailored to code energy performance.

Furthermore, GreenCode provides an ideal framework for the empirical evaluation and refinement of existing sustainable programming standards[21]. By rigorously testing established green software guidelines, the project aims to contribute to their evolution, ensuring that best practices are validated against real-world data and enriched through continuous iterative improvement.

[21] https://sci.greensoftware.foundation/

2.3 Benchmark Analysis

2.3.1 Background

The increasing complexity and scale of software systems, coupled with the rise of Large Language Models (LLMs) for code generation, has intensified the need for sustainable and efficient software engineering practices. Within this context, Software Engineering advocates optimizing software systems to improve code efficiency and resource efficiency by energy efficiency, reduce runtime and memory usage, and extend hardware longevity[22,23]. However, while LLMs can automate code generation and optimization, their efficiency-aware behaviour remains underexplored. Evaluating such efficiency requires stable benchmarks and standardized measurement methods.

Recent studies have introduced a new generation of LLM code generation and optimization benchmarks, each addressing different aspects of performance, correctness, and energy awareness. these efforts collectively establish a foundation for systematic, efficiency-oriented evaluation of LLMs, enabling measurable progress toward sustainable and energy-aware software development[24].

2.3.2 Current State of the Art

Recent advances in benchmarking for Large Language Model (LLM) code generation and optimisation have produced a diverse but fragmented landscape. Most benchmarks focus on evaluating LLMs' ability to generate efficient and sustainable code, with particular emphasis on function-level tasks due to their simplicity and reproducibility. Python is the predominant language, though some benchmarks extend to C/C++, Java, Verilog, JavaScript, Ruby, and Go. Evaluation criteria are largely centred on runtime and memory efficiency, with some benchmarks incorporating qualitative metrics such as readability and maintainability, and a few considering hardware-level performance. There is a growing trend towards multi-dimensional and developer-centric metrics, and some frameworks now support more complex, program-level or repository-level evaluations. However, energy consumption and broader sustainability metrics remain largely absent. A recent study by Castaño et al. revealed that for 99% of the Hugging Face ML models (N>170.000), for about 200 model only key-emissions-related context is reported, for about 1350 models carbon emission is reported without context or optimization, for about 75 models emission data is provided together with the related context, and zero models met certified energy efficiency standards[25]. Furthermore, benchmarking practices and environments are inconsistent, limiting comparability and reproducibility.

2.3.3 Key Findings

- Dominance of Function-Level Benchmarks: Most studies assess LLMs at the function level, which enables controlled experiments but does not reflect real-world complexity.
- Python as the Central Language: Most benchmarks use Python, underlining its significance in LLM code research.

[22] SWE-Perf: Can Language Models Optimize Code Performance on Real-World Repositories?
[23] Mingzhe Du, Luu Anh Tuan, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: a code efficiency benchmark for code large language models. In Proceedings of the 38th International Conference on Neural Information Processing Systems (Vancouver, BC, Canada) (NIPS '24).
[24] Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. 2023. Estimating the carbon footprint of BLOOM, a 176B parameter language model. J. Mach. Learn. Res. 24, 1, Article 253 (January 2023), 15 pages.
[25] J. Castaño, S. Martínez-Fernández, X. Franch and J. Bogner, "Exploring the Carbon Footprint of Hugging Face's ML Models: A Repository Mining Study," *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, New Orleans, LA, USA, 2023, pp. 1-12, doi: 10.1109/ESEM56168.2023.10304801.

- Diverse Evaluation Metrics: While runtime and memory usage are the primary metrics, some benchmarks (e.g., RACE[26], CodeEditorBench[27]) also consider readability, maintainability, and complexity.
- Hardware and Platform Awareness: Benchmarks like EvalPerf[28], Coffe[29], and ResBench[30] employ hardware counters and instruction-based measures for reproducibility.
- Emergence of Realistic Benchmarks: SWE-Perf[31] and MARCO[32] represent efforts to benchmark LLMs using full codebases and realistic execution environments.
- Lack of Direct Energy Measurement: No benchmarks currently provide standardised, direct measures of energy consumption or carbon impact, relying instead on proxies like runtime or hardware counters.

2.3.4 Identified Gaps

- Limited Realism and Granularity: Most benchmarks do not assess program- or repository-level tasks, missing out on real-world software system complexity.
- Absence of Sustainability Metrics: There is a notable lack of direct evaluation for energy efficiency, power usage, and environmental impact.
- Data Contamination: Overlap between benchmark datasets and LLM training corpora risks bias and overestimation of LLM performance.
- Inconsistent Experimental Setups: Variability in hardware and non-standardised environments undermines reproducibility and comparability.
- Narrow Evaluation Focus: Benchmarks predominantly target correctness and performance, often neglecting maintainability, readability, and robustness – qualities essential for sustainable code.

Overall, while the benchmarking ecosystem for LLM-driven code generation is advancing, it remains fragmented and primarily focused on computational performance. There is a pressing need for standardised, reproducible benchmarks that integrate sustainability, environmental impact, and real-world software engineering objectives.

2.3.5 Summary and Future Opportunities for GreenCode

The review of existing LLM-based benchmarks reveals a growing yet fragmented ecosystem. Most studies emphasize function-level efficiency and measure performance primarily through runtime and memory metrics. While these offer a solid foundation for evaluating computational performance, they neglect broader aspects such as energy efficiency and environmental impact.

[26] Zheng, J., Cao, B., Ma, Z., Pan, R., Lin, H., Lu, Y., ... & Sun, L. (2024). Beyond correctness: Benchmarking multi-dimensional code generation for large language models. *arXiv preprint arXiv:2407.11470*.
[27] Jiawei Guo et al.; CodeEditorBench: Evaluating Code Editing Capability of LLMs; Proceedings of ICLR 2025 Third Workshop on Deep Learning for Code; 2025; https://openreview.net/forum?id=6yTgoh0J0X
[28] Liu, J., Xie, S., Wang, J., Wei, Y., Ding, Y., & Zhang, L. (2024). Evaluating language models for efficient code generation. *arXiv preprint arXiv:2408.06450*.
[29] Peng, Y., Wan, J., Li, Y., & Ren, X. (2025). Coffe: A code efficiency benchmark for code generation. *Proceedings of the ACM on Software Engineering*, *2*(FSE), 242-265.
[30] Guo, C., & Zhao, T. (2025). Resbench: Benchmarking llm-generated fpga designs with resource awareness. *arXiv preprint arXiv:2503.08823*.
[31] He, X., Liu, Q., Du, M., Yan, L., Fan, Z., Huang, Y., ... & Ma, Z. (2025). Swe-perf: Can language models optimize code performance on real-world repositories?. *arXiv preprint arXiv:2507.12415*.
[32] Asif Rahman, Veljko Cvetkovic, Kathleen Reece, Aidan Walters, Yasir Hassan, Aneesh Tummeti, Bryan Torres, Denise Cooney, Margaret Ellis, and Dimitrios S. Nikolopoulos. [n. d.]. Performance Evaluation of Large Language Models for High-Performance Code Generation: A Multi-Agent Approach (MARCO). https://api.semanticscholar.org/CorpusID:280547604

Evaluation practices remain inconsistent, and direct energy measurements are rarely included. LLM integration patterns are also limited, dominated by static, single-turn text-to-code generation, with few frameworks supporting iterative or agentic refinement.

Overall, current benchmarks capture only a partial view of software efficiency. Future efforts within GreenCode should build upon these findings by developing standardized, reproducible, and energy-aware benchmarks that link performance with sustainability. This direction will strengthen the methodological basis for evaluating LLM-generated code and sustainable software engineering practices.

2.4 Quality Assessment of GenAI outcomes

2.4.1 Background

The evaluation of generative AI (GenAI) outputs has emerged as a pivotal concern across multiple domains, including computer science, healthcare[33], law[34], and, notably, software engineering[35]. Recent research highlights that quality in GenAI is inherently multidimensional, encompassing factual accuracy, completeness, reasoning coherence, clarity, style, safety, and trustworthiness. In technical fields such as software engineering and automotive systems, these criteria expand further to address domain-specific requirements, including compliance with safety standards and real-world hardware constraints. This complexity underscores the necessity for tailored evaluation frameworks capable of capturing the full breadth of GenAI output quality. Full version can be found in the annex.

2.4.2 Current State of the Art

Traditional evaluation metrics, such as BLEU and ROUGE, are increasingly recognised as insufficient for assessing open-ended or creative GenAI tasks. These metrics primarily reward textual similarity, failing to account for faithfulness, usefulness, or deeper semantic qualities. To address these shortcomings, the field has seen a shift towards "LLM-as-a-judge"[36] methodologies, wherein powerful language models are employed to assess the outputs of other models. Frameworks like G-Eval[37] have demonstrated improved alignment with human judgements regarding factuality and coherence, though concerns remain regarding evaluator bias and self-agreement.

In software engineering, evaluation now extends beyond fluency and task completion to include maintainability, security vulnerabilities, and static-analysis defects. Notable domain-specific frameworks such as CODEJUDGE[38] leverage large language models to assess semantic correctness, moving beyond reliance on test cases alone. Similarly, benchmarks like L2CEval[39] focus on calibration and error analysis in code generation, revealing that correctness is just one

[33] Daniel Rodger, Sebastian Porsdam Mann, Brian Earp, Julian Savulescu, Christopher Bobier, Bruce P. Blackshaw,Generative AI in healthcare education: How AI literacy gaps could compromise learning and patient safety,Nurse Education in Practice,Volume 87,2025,104461,ISSN 1471-5953, https://doi.org/10.1016/j.nepr.2025.104461.

[34] https://www.theguardian.com/us-news/2025/may/31/utah-lawyer-chatgpt-ai-court-brief

[35] Shukla, Shubham, Gen AI for Code Vulnerability Detection and Risk Analysis (December 02, 2023). Available at SSRN: https://ssrn.com/abstract=5173531 or http://dx.doi.org/10.2139/ssrn.5173531

[36] Gu, J., et al. "A Survey on LLM-as-a-Judge", <i>arXiv e-prints</i>, Art. no. arXiv:2411.15594, 2024. doi:10.48550/arXiv.2411.15594.

[37] https://deepeval.com/docs/metrics-llm-evals

[38] https://github.com/VichyTong/CodeJudge

[39] Ansong Ni, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, Shafiq Joty, Yingbo Zhou, Dragomir Radev, Arman Cohan, Arman Cohan; L2CEval: Evaluating Language-to-Code Generation Capabilities of Large Language Models. Transactions of the Association for Computational Linguistics 2024; 12 1311–1329. doi: https://doi.org/10.1162/tacl_a_00705

aspect of overall quality. Static code analysis tools are increasingly used to evaluate maintainability and reliability, identifying issues such as code smells and security flaws in GenAI-generated artefacts.

2.4.3 Key Findings

- There is a weak correlation between functional correctness and code quality in GenAI-generated software; code that passes functional tests may still harbour maintainability or reliability issues.
- Persistent hallucination – defined as the confident generation of false or unsupported content – remains a significant quality defect, with empirical audits revealing ongoing challenges across models and domains.
- Domain-specific advances, particularly in safety-critical areas like automotive software, demonstrate the viability of integrating GenAI evaluation with formal verification, requirements analysis, and system-level validation.
- Novel frameworks and empirical studies increasingly recognise the need to evaluate GenAI outputs against a broader set of criteria, including security, integration correctness, and compliance with domain standards.

2.4.4 Identified Gaps

- Surface-similarity metrics (e.g., BLEU[40], ROUGE[41]) are inadequate for capturing the full spectrum of GenAI output quality, particularly regarding semantic faithfulness and utility.
- Evaluator bias and self-agreement in LLM-based evaluation frameworks present ongoing challenges to objective assessment.
- There is a lack of comprehensive, standardised benchmarks that holistically assess GenAI outputs across multiple quality dimensions, including sustainability and energy efficiency.
- Integration of energy and sustainability metrics into GenAI evaluation remains limited, despite growing recognition of their importance.

2.4.5 Future Opportunities for GreenCode

Building on these findings, GreenCode is well-positioned to advance the field through the following targeted actions:

- Develop standardised, reproducible, and energy-aware benchmarks that connect GenAI performance with sustainability objectives.
- Integrate advanced, multidimensional quality metrics – encompassing maintainability, security, and domain compliance – into evaluation frameworks for GenAI-generated code.
- Address hallucination detection and calibration by adopting uncertainty estimation approaches and entropy-based confidence measures to flag unreliable outputs.
- Expand and refine domain-specific evaluation frameworks, particularly for safety-critical and embedded systems, to ensure trustworthy and robust GenAI integration in software engineering workflows.

[40] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu; BLEU: a Method for Automatic Evaluation of Machine Translation; Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL), Philadelphia, July 2002, pp. 311-318; https://aclanthology.org/P02-1040.pdf

[41] Chin-Yew Lin; ROUGE : A Package for Automatic Evaluation of Summaries; In Proceedings of Workshop on Text Summarization Branches Out, Post-Conference Workshop of ACL 2004, Barcelona, Spain. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/was2004.pdf

By pursuing these opportunities, GreenCode can establish itself at the forefront of sustainable, high-quality GenAI evaluation in software engineering, thereby strengthening both methodological rigour and practical impact.

2.5 Static code analysers for compiled software

2.5.1 Background

Static code analysis is the compile time analysis of an application to verify certain aspects of the code. It is typically applied to verify certain correctness criteria like the absence of run-time errors, the protection of sensitive information or the functional correctness of the code.

Static analysis can, however, also be applied to trace resource usage like CPU time or memory usage. This provides a compile-time mechanism to estimate or even limit the resources required by an application.

2.5.2 Current State of the Art

Static code analysis methods are broadly grouped into three categories: formal verification (using tools like Rocq[42] or Isabelle[43]), model checking, and abstract interpretation. Formal verification relies on developers to provide detailed annotations (such as pre- and postconditions) that define the behaviour to be verified, but the process is often too labour-intensive for practical use. Model checkers systematically examine the possible states a program can reach to determine, for instance, whether error states are accessible; however, this approach struggles with the vast state spaces found in real-world software, making it frequently impractical. Abstract interpretation, meanwhile, simulates program execution using abstract values to cover all potential execution paths and values, typically without needing developer input. It employs a fixed-point algorithm to analyse data flow until stable conditions are met but may still suffer from state explosion depending on the abstraction's granularity.

2.5.3 Key Findings

- Formal verification provides strong guarantees but requires significant developer effort for annotation, limiting its practical adoption.
- Model checking can offer rigorous state-space analysis, but its applicability is curtailed by the exponential growth of possible program states in complex systems.
- Abstract interpretation automates analysis and does not depend on developer annotations, making it more scalable, but it risks state explosion when value representations are too detailed.

2.5.4 Limitations and Gaps

Static code analysis is typically limited to the verification of functional correctness and does not trace any metrics related to resource usage.

Formal verification and Model checking in general cannot be easily applied to arbitrary code. Abstract interpretation, however, may be used to analysis whole applications.

In a system that uses effects to model resource usage, effect handler values could be used to analyse, trace and verify resource constraints. A static analyser using abstract interpretation that represents values of effects that model resources could provide a means to fill this gap.

[42] Rocq Prover (2025), https://rocq-prover.org/

[43] Isabelle Contributors: Isabelle proof assitant. https://isabelle.in.tum.de/

2.5.5 Summary and Future Opportunities for GreenCode

Abstract Interpretation used to trace effect handlers that model resource usage could provide a means to statically analyse and manage the resource usage of application code.

3 Conclusion and Synthesis

The GreenCode partners examined the strengths and limitations of static code analysis techniques – specifically formal verification, model checking, and abstract interpretation – with a focus on their applicability to managing and verifying resource usage in software. The state-of-the-art reviews identify that formal verification and model checking, while rigorous, struggle with scalability and are not easily applicable to arbitrary code due to the exponential growth of program states. Abstract interpretation, on the other hand, is more scalable and does not require developer annotations, but may also encounter state explosion if value representations become overly detailed. Importantly, conventional static analysis primarily addresses functional correctness and often overlooks resource usage metrics.

A notable gap in current methods is the inability to statically trace or verify resource consumption directly within application code. The document highlights a promising opportunity: leveraging effect handlers and abstract interpretation to model and analyse resource usage. By representing effect values that correspond to resource consumption, it becomes feasible to statically analyse, trace, and even enforce resource constraints within software. This approach could significantly improve the static management of energy and other resources, directly addressing the shortcomings of traditional methods.

To further inform this direction, a literature review was also conducted to explore existing approaches for predicting software energy consumption through static analysis. The review was driven by targeted research questions, robust inclusion and exclusion criteria, and a hybrid search methodology combining Scopus database queries with backward and forward snowballing. This comprehensive strategy ensured a thorough identification and assessment of relevant publications, thus mapping the current landscape of software energy prediction techniques. Another notable limitation identified is the absence of robust, widely accepted benchmarks for evaluating and comparing the effectiveness of these energy prediction methods. The lack of standardised benchmarks impedes the ability to consistently measure tool performance, validate results, and drive reproducible research in this area. Addressing this gap will be crucial for future work aiming to advance the reliability and comparability of static energy analysis approaches.

For the GreenCode project, the key opportunity lies in advancing static analysis by integrating effect handlers and abstract interpretation to enable more precise and scalable resource usage analysis. This not only fills a critical gap in the state of the art but also positions GreenCode to contribute novel solutions for the static management of energy and resource consumption within software applications.

4 Annex

4.1 Mapping of software system's codebase

To address the gaps identified in the analysis of the state of the art, GreenCode adopts a state-of-the-art, parser-driven methodology based on Model-Driven Reverse Engineering (MDRE) using ANTLR4. This approach:

- ➢ Automates the extraction of detailed, formal models from source code, enabling iterative optimization.
- ➢ Ensures accuracy by relying on formal grammar for parsing, reducing errors and ambiguities.
- ➢ Produces standardized, machine-readable outputs (e.g., XMI), facilitating interoperability with other engineering tools.
- ➢ Implements the SACA module, which operationalizes this methodology for multiple programming languages, generating comprehensive, structured maps of codebases as the foundation for further quality assurance and AI-driven optimization.

4.1.1 The ANTLR4 Methodology for Automated Model Extraction

The core of the GreenCode proposed mapping strategy is an ANTLR4-driven MDRE methodology, as presented in Khachouch et al.[44]. ANTLR4 (ANother Tool for Language Recognition) is a parser generator that can process source code based on a formal language grammar. The methodology leverages an ANTLR4-generated "visitor" pattern to programmatically traverse the parse tree (or syntax tree) created from the source code. As the visitor encounters each grammatical construct – such as a class declaration, a method definition, or a variable assignment – it systematically instantiates a corresponding element in a target metamodel, such as the Knowledge Discovery Metamodel (KDM). The process culminates in the serialization of this model into a standardized format like XMI (XML Metadata Interchange), making it machine-readable and interoperable.

This ANTLR4-driven MDRE approach offers several key advantages for the GreenCode pipeline, namely:

- ➢ Accuracy: It addresses the challenge of correctly parsing complex source languages by relying on formal, unambiguous grammars, ensuring a precise interpretation of the code.
- ➢ Automation: It streamlines the reverse engineering workflow by automating the transformation from raw source code to a structured model, eliminating the need for manual and error-prone interpretation.
- ➢ Fidelity: It enables the generation of faithful target models that accurately represent the original system's structure and semantics, providing a reliable foundation for subsequent analysis.
- ➢ Standardization: It facilitates the creation of models in standardized formats, such as XMI (XML Metadata Interchange), ensuring interoperability with a wide range of model-driven engineering tools and platforms.

4.1.2 Practical Implementation: The SACA Module

This academically validated methodology is being put into practice within the GreenCode project through the SACA module (Figure 1). SACA is the specific component responsible for executing

[44] M.K. Khachouch, A. Korchi, M. Bekkali, Y. Lakhrissi (2024) "*ANTLR4-Driven Model-Driven Reverse Engineering: Bridging Source Language Parsing and Metamodel Instantiation*", Journal of Logistics, Informatics and Service Science, Volume 11, Issue 11, pp. 426-446, ISSN 2409-2665, https://doi.org/10.33168/JLISS.2024.1123.

the detailed codebase mapping. It operationalizes the ANTLR4-driven approach to produce the foundational codebase model for the GreenCode pipeline. For its part, ANTLR4 is a powerful and flexible parser generator that is integral to the development of compilers and language processing tools. Its proven ability to systematically parse complex source languages makes it an ideal foundation for building a precise, scalable, and automated codebase mapping tool that can meet the project's static analysis needs.

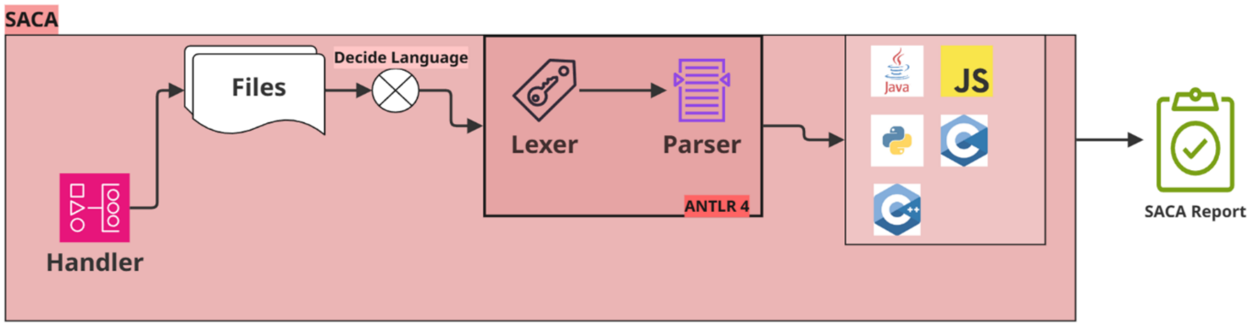


Figure 1. SACA Module Architecture

The ANTLR4-driven process transforms unstructured source code into a detailed, hierarchical representation. This transformation occurs in a sequence of well-defined steps:

- Language Grammars: The process begins with a formal grammar file (with a .g4 extension) for each programming language to be analyzed. These grammars define the language's syntax rules and are often sourced from established public repositories, such as https://github.com/antlr/grammars-v4.
- Lexer (Tokenization): The Lexer scans the raw source code and breaks it into a sequence of categorized tokens. Each token represents a piece of the code, such as a keyword (if), an identifier (my_variable), or an operator (+).
- Parser and Parse Tree Construction: The Parser consumes the stream of tokens generated by the Lexer to construct a Parse Tree, also known as a Concrete Syntax Tree (CST). This tree is a detailed, hierarchical representation of the code's grammatical structure, mirroring the rules defined in the language grammar.
- Listeners and Visitors: To extract meaningful information, a Listener or Visitor pattern is used to "walk" the parse tree. As the listener enters and exits specific nodes in the tree (e.g., a classDeclaration or methodDeclaration), custom logic is executed to identify and count features or instantiate elements in a target metamodel.

The key features of the SACA module are summarized in Table 1:

Table 1. Key features of the SACA module

Feature	Description
Core Technology	Implements the ANTLR 4 library in Python.
Supported Languages	Designed to analyze and map source code for Python, C, C++, JavaScript, and Java.
Core Logic	A Handler class scans a folder of source code, identifies the language of each file, and dynamically selects the appropriate ANTLR 4 Lexer and Parser.
Primary Output	A JSON report detailing the features found in each analysed file, creating a machine-readable index of the codebase.
Mapped Features	Identifies and counts structural elements such as the number of classes, methods, statements, and keywords.

This detailed, structured map of the codebase serves as the foundational input for all subsequent stages within the GreenCode pipeline, including advanced quality assurance, infrastructure assessment, and the AI-driven optimization cycles that are central to the project's goals.

4.2 Static code analysis with focus on energy

To learn about existing approaches for predicting software energy consumption based on the static analysis of software code, we performed systematic literature review. The review was guided by the following research questions (RQ):

- RQ1: What kind of software is considered?
- RQ2: What features are considered for energy prediction?
- RQ3: What is the output of the energy prediction?
- RQ4: What type of models are used to make the predictions?
- RQ5: How and with what tools is the actual energy consumption measured?
- RQ6: What environment is the software operating in and how it is configured?

To identify relevant publications, we first selected a literature database and defined an appropriate search strategy. We used the Scopus database[45]. To reduce the risk of missing relevant work, we performed backward and forward snowballing[46] starting from seed papers identified via the Scopus search – a hybrid approach we denote as "Scopus + BS*FS"[47]. Based on the research questions, we constructed the search query (see Table 2) to identify potentially relevant publications, which we first screened for relevance by reviewing their title and abstract. To assess the relevant of the publication we used predefined inclusion and exclusion criteria (see Table 3). We reviewed the remaining relevant publications in full text and extracted data necessary for answering the research questions. We took the relevant papers as seed for snowballing, which identified additional relevant publications. Figure 2 summarizes the entire review process and the number of publications resulting from each step.

Table 2. Search query

```
TITLE-ABS-KEY (
   (((software OR code) PRE/2 (energy OR power) PRE/2 (consumption OR usage))
   AND (predict* OR estimat* OR forecast* OR model*)) OR
   ((static PRE/1 analysis) AND ((energy OR power) PRE/2 (consumption OR usage))
   AND (predict* OR estimat* OR forecast* OR model*)))
   AND
TITLE (
   (software OR code OR application*) AND (energy OR power) AND (consumption OR
usage) AND
   (predict* OR estimat* OR forecast* OR model*))
   AND
PUBYEAR > 2014 AND PUBYEAR < 2026 AND (LIMIT-TO (DOCTYPE,"cp") OR LIMIT-TO
(DOCTYPE,"ar"))
AND (LIMIT-TO (SUBJAREA,"COMP") OR LIMIT-TO (SUBJAREA,"ENGI") OR LIMIT-TO
(SUBJAREA,"MATH")
OR LIMIT-TO (SUBJAREA,"ENER")) AND (LIMIT-TO ( LANGUAGE,"English"))
```

[45] https://www.scopus.com/

[46] Claes Wohlin, Marcos Kalinowski, Katia Romero Felizardo, Emilia Mendes; Successful combination of database search and snowballing for identification of primary studies in systematic literature studies, Information and Software Technology, Volume 147, 2022, 106908, ISSN 0950-5849, https://doi.org/10.1016/j.infsof.2022.106908.

[47] Erica Mour˜ao et al. "On the performance of hybrid search strategies for systematic literature reviews in software engineering". In: Information and Software Technology 123 (July 2020), p. 106294. issn: 0950-5849. doi: 10.1016/j.infsof.2020.106294. url: http://dx.doi.org/10.1016/j.infsof.2020.106294.

Table 3. Inclusion and exclusion criteria

Criteria	Description
Inclusion (IC1)	The work concerns predicting software energy consumption based on the source code.
Exclusion (EC1)	The work does not consider a model, e.g., statistical or machine learning, for the predicting energy consumption.
Exclusion (EC2)	The work is not accessible.
Exclusion (EC3)	The work is a secondary study, i.e., survey, systematic literature review or systematic mapping study.
Exclusion (EC4)	The work has been published before 2015 or after 11.09.2025[48].
Exclusion (EC5)	The work is not written in English.
Exclusion (EC6)	The work is grey literature.

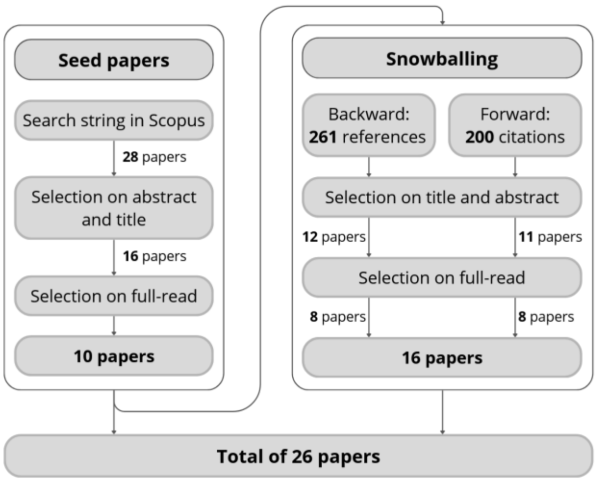


Figure 2. Systematic literature review process

The following sections present the results of the literature survey according the the research questions.

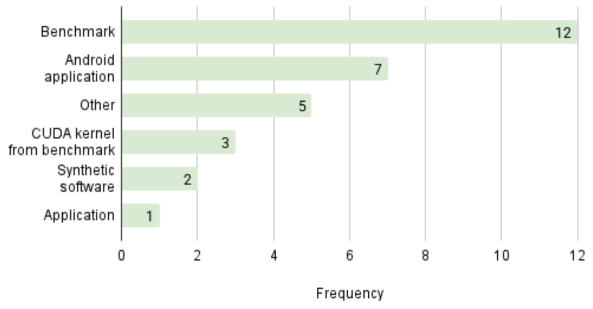
➢ RQ1: What kind of software is considered?

The software analysed in the reviewed studies covers a wide range sources, types and languages. Figure 3 shows that the most common type of software originates from benchmark suites (12 studies), including AnghaBench[49], BEEBS[50], PolyBench[51], Rodinia[52], and SPEC CPU2006[53]. Two of these 12 studies combine synthetic software for training with benchmark programs for testing[54,55].

[48] On this date the literature database search was closed.
[49] A. Faustino da Silva et al. „ANGHABENCH: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction". In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2021, pp. 378-390. doi: 10.1109/CGO51591.2021.9370322.
[50] J. Pallister, S. Hollis, and J. Bennett. BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms. 2013. arXiv: 1308.5174 [cs.PF]. url: https://arxiv.org/abs/1308.5174.
[51] Pouchet Louis-Noel. Polybench: The polyhedral benchmark suite. https://www.cs.colostate.edu /~pouchet/software/polybench/. [Accessed: 2025-10-13]. 2012.
[52] Shuai Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: 2009 IEEE International Symposium on Workload Characterization (IISWC). 2009, pp. 44-54. doi: 10.1109/IISWC.2009.5306797.
[53] Standard Performance Evaluation Corporation. SPEC CPU 2006 benchmark (Retired: January 2018). https://www.spec.org/cpu2006/. [Accessed: 2025-10-13].
[54] Charalampos Marantos, Nikolaos Maidonis, and Dimitrios Soudris. "Designing Application Analysis Tools for Cross-Device Energy Consumption Estimation". In: 2022 11th International Conference on Modern Circuits and Systems Technologies (MOCAST). 2022, pp. 1–4. doi: 10.1109/MOCAST54814.2022.9837632.
[55] Charalampos Marantos et al. "A Flexible Tool for Estimating Applications Performance and Energy

Three additional studies also use benchmarks but target GPU-intensive tasks (CUDA kernels), so we classify them separately[56, 57, 58]. The second-largest group analyses Android applications from platforms such as AndroZoo[59] and GreenOracle[60, 61]. In Zhang et al.[62], the applications run on a Linux computer rather than an Android device. Finally, we group various programs and embedded software under Other proprietary software.

Regarding programming languages, C is the most frequently used, often combined with C++ for GPU or desktop/server programs (see the right chart in Figure 3). CUDA is used for GPU kernels, while Java appears mainly in mobile or synthetic program scenarios. Python appears in only one paper[63], alongside other languages and without a specific focus. Fortran is used occasionally for legacy benchmarks or scientific computing. Additionally, 9 studies do not specify the programming language; most of these focus on mobile applications where user behaviour, rather than source code, is the primary concern.



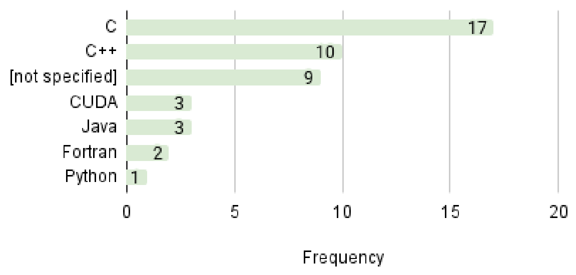


Figure 3. Type of software and programming languages considered in the related literature

- RQ2: What features are considered for energy prediction?

When it comes to the input of the prediction models, we have categorized it in two ways: input source and granularity. According to Hoque et. al.[64] there are three types of input and the associated

Consumption Through Static Analysis". In: SN Comput. Sci. 2.1 (Jan. 2021). doi: 10.1007/s42979-020-00405-7. url: https://doi.org/10.1007/s42979-020-00405-7.

56 Gargi Alavani Prabhu et al. " Estimating Power Consumption of GPU Application Using Machine Learning Tool ". In: 2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI). Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2024, pp. 734–739. doi: 10.1109/ICTAI62512. 2024.00109. url: https://doi.ieeecomputersociety.org/10.1109/ICTAI62512.2024.00109.

57 Gargi Alavani et al. "Program Analysis and Machine Learning–based Approach to Predict Power Consumption of CUDA Kernel". In: ACM Trans. Model. Perform. Eval. Comput. Syst. 8.4 (July 2023). issn:2376-3639. doi: 10.1145/3603533. url: https://doi.org/10.1145/3603533.

58 Lorenz Braun et al. A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels. 2020. arXiv: 2001.07104 [cs.DC]. url: https://arxiv.org/abs/2001.07104.

59 Shaiful Chowdhury et al. "Greenscaler: training software energy models with automatic test generation". In: Empirical Software Engineering 24.4 (2019), pp. 1649–1692.

60 Stephen Romansky et al. "Deep green: Modelling time-series of software energy consumption". In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE. 2017, pp. 273–283.

61 Shaiful Alam Chowdhury and Abram Hindle. "Greenoracle: Estimating software energy consumption with energy measurement corpora". In: Proceedings of the 13th international conference on mining software repositories. 2016, pp. 49–60.

62 Tong Zhang et al. "Assessing Predictive Models for Energy Consumption Across Varied Software Environments". In: 2024 IEEE International Conference on Big Data (BigData). Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2024, pp. 5233–5242. doi: 10.1109/BigData62323.2024.10825500. url: https://doi.ieeecomputersociety.org/10.1109/BigData62323.2024.10825500.

63 Cuijiao Fu, Depei Qian, and Zhongzhi Luan. "Estimating software energy consumption with machine learning approach by software performance feature". In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). IEEE. 2018, pp. 490–496.

64 Mohammad Ashraful Hoque et al. "Modeling, Profiling, and Debugging the Energy Consumption of

three types of models, where the input's type determines the model's type. This classification has been widely used in related secondary studies[65, 66] and it includes:

- Utilization-based Models: These correlate the energy usage of hardware components with their resource utilization. Typical inputs include CPU, memory, and cache usage metrics, often collected through hardware performance counters (HPCs). Such models assume that power consumption is proportional to the internal activity of the processor.
  - Examples in the related literature include: Cellular utilization, CPU utilization, Storage Utilization, GPU utilization, Display utilization and Network utilization. Cellular utilization, for example, refers to the monitoring of hardware components in mobile devices (e.g., CPU frequency, screen, Wi-Fi, Bluetooth).
- Event-based Models: These rely on system events (e.g., system calls or network protocol state transitions) to characterize power consumption, especially when hardware components exhibit non-linear or "tail" energy behavior. By tracking events like read/write calls or radio state changes, they provide more accurate estimates in dynamic contexts.
  - Examples seen in the literature: Performance Monitoring Counter (PMC) and System calls. PMC are hardware-level metrics that record low-level processor activities such as cache misses, instructions per cycle, or context switches, providing detailed insights into the relationship between system performance and energy usage[67]. System calls represent interactions between an application and the operating system, capturing high-level software behavior that helps identify how programs use system resources and thus influence energy consumption.
- Code-analysis-based Models: These estimate energy consumption statically, by analyzing program code without execution. They often work at instruction or function level, associating each operation with an estimated power cost. While useful early in development, they are less accurate for context-dependent behaviors, such as network conditions.
  - Examples seen in the literature: Parallel Thread Execution (PTX) features, Opcode counts, Low Level Virtual Machine (LLVM) features, and software metrics. PTX (Parallel Thread Execution) can be thought as the assembly language of the NVIDIA CUDA GPU computing platform[68]. From this representation the studies extract features following

Mobile Devices". In: ACM Comput. Surv. 48.3 (Dec. 2015). issn: 0360-0300. doi: 10.1145/2840723. url: https://doi.org/10.1145/2840723.

[65] Andreas Schuler and Gabriele Kotsis. "A systematic review on techniques and approaches to estimate mobile software energy consumption". In: Sustainable Computing: Informatics and Systems 41 (2024), p. 100919. issn: 2210-5379. doi: https://doi.org/10.1016/j.suscom.2023.100919. url: https: //www.sciencedirect.com/science/article/pii/S2210537923000744.

[66] Raja Wasim Ahmad et al. "A Review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues". In: Journal of Network and Computer Applications 58 (2015), pp. 42–59. issn: 1084-8045. doi: https://doi.org/10.1016/j.jnca.2015.09.002. url: https://www.sciencedirect. com/science/article/pii/S1084804515002088.

[67] Tong Zhang et al. "Assessing Predictive Models for Energy Consumption Across Varied Software Environments". In: 2024 IEEE International Conference on Big Data (BigData). Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2024, pp. 5233–5242. doi: 10.1109/BigData62323.2024.10825500. url: https://doi.ieeecomputersociety.org/10.1109/BigData62323.2024.10825500.

[68] Scudiero, Tony and Bentz, Jonathan. Understanding PTX, the Assembly Language of CUDA GPU Computing.
https://developer.nvidia.com/blog/understanding-ptx-the-assembly-language-of-cudagpu-computing/. [Accessed: 2025-10-23].

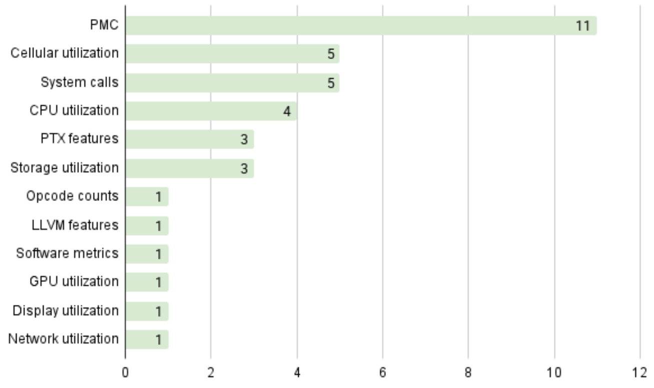
different approaches[69, 70, 71]. LLVM (Low Level Virtual Machine) is a compiler framework that provides an intermediate representation of code[72]. This approach is only followed once by Marantos et. al.[73], and they use LLVM Machine Code Analyzer to extract the desired features.

The left side of Figure 4 shows the frequency of each input source. PMC is the most used input type (11); followed by Cellular utilization and System calls (5); CPU utilization is being used 4 times; PTX features and Storage utilization are being used 3 times; and lastly, Opcode counts, LLVM features, Software metrics, GPU utilization, Display and Network utilization, are only used once. Overall, there is a clear tendency to use event-based input.

Regarding the granularity, we describe it as the grain that the model is predicting energy of. From biggest to smallest there is:

- Application: Energy consumption is predicted for an entire program.
- Kernel or function: Energy consumption is predicted for a kernel or function, whereas Kernel corresponds to a function and is used specifically for GPU kernels.
- Basic Block: Energy consumption is predicted for a segment of a program.
- Sample: Energy consumption is predicted for a timestamp throughout time interval (i.e., time-series prediction).

Figure 4, plot on the right, shows, most approaches predict the EC for an entire program (17); 14 studies at application-level and 3 studies at kernel-level. Followed by 5 studies that predict it for basic blocks and 4 studies that do it per sample or time-stamp.



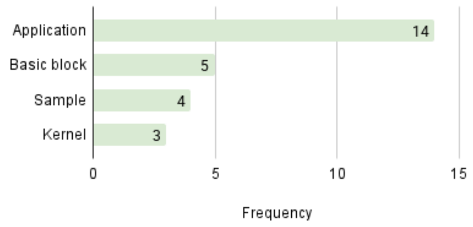


Figure 4. Prediction method (left) and the granularity of prediction (right)

[69] Gargi Alavani Prabhu et al. “Estimating Power Consumption of GPU Application Using Machine Learning Tool ”. In: 2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI). Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2024, pp. 734–739. doi: 10.1109/ICTAI62512. 2024.00109. url: https://doi.ieeecomputersociety.org/10.1109/ICTAI62512.2024.00109.

[70] Gargi Alavani et al. “Program Analysis and Machine Learning–based Approach to Predict Power Consumption of CUDA Kernel”. In: ACM Trans. Model. Perform. Eval. Comput. Syst. 8.4 (July 2023). issn: 2376-3639. doi: 10.1145/3603533. url: https://doi.org/10.1145/3603533.

[71] Lorenz Braun et al. A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels. 2020. arXiv: 2001.07104 [cs.DC]. url: https://arxiv.org/abs/2001. 07104.

[72] LLVM. The LLVM Compiler Infrastructure. https://llvm.org/. [Accessed: 2025-10-22].

[73] Charalampos Marantos et al. “A Flexible Tool for Estimating Applications Performance and Energy Consumption Through Static Analysis”. In: SN Comput. Sci. 2.1 (Jan. 2021). doi: 10.1007/s42979-020-00405-7. url: https://doi.org/10.1007/s42979-020-00405-7.

➢ RQ3: What is the output of the energy prediction?

Model outputs are primarily *energy* or *power* (Figure 5). Power is the most common (16 studies); energy is second (10). A smaller subset also predicts performance metrics, such as execution time[74, 75] or instructions per second[76]. Reymond et al.[77] and Wegener et al.[78] target worst-case metrics – worst-case execution time (WCET) and worst-case energy consumption (WCEC) – which are especially relevant in embedded and real-time systems; we group these under time and energy, respectively. Of the six studies with multiple target variables, four combine time with energy or power, and two combine power with performance.

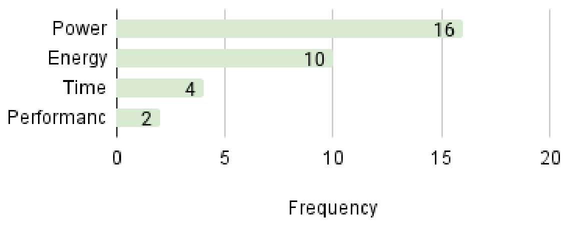


Figure 5. Output of energy prediction models

➢ RQ4: What type of models are used to make the predictions?

The predictive models employed in the surveyed studies (Figure 6) can be broadly categorized into five groups: linear models (LMs), tree-based models (TBMs), neural networks (NNs), kernel-based models (KBMs), and distance-based models (DBMs). Linear models remain a common baseline in almost all studies, with variants such as Ordinary Least Squares, Linear Regression, Lasso, Ridge, Elastic Net, and Bayesian Ridge being frequently adopted. These approaches are valued for their interpretability and low computational overhead, making them suitable for fast prediction and comparative baselines.

[74] Lorenz Braun et al. A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels. 2020. arXiv: 2001.07104 [cs.DC]. url: https://arxiv.org/abs/2001.07104.
[75] Charalampos Marantos et al. "A Flexible Tool for Estimating Applications Performance and Energy Consumption Through Static Analysis". In: SN Comput. Sci. 2.1 (Jan. 2021). doi: 10.1007/s42979-020-00405-7. url: https://doi.org/10.1007/s42979-020-00405-7.
[76] Shivam Kundan, Ourania Spantidi, and Iraklis Anagnostopoulos. "Online frequency-based performance and power estimation for clustered multi-processor systems". In: Computers Electrical Engineering 90 (Mar. 2021), p. 106971. doi: 10.1016/j.compeleceng.2021.106971.
[77] Hugo Reymond, Abderaouf Nassim Amalou, and Isabelle Puaut. "WORTEX: Worst-Case Execution Time and Energy Estimation in Low-Power Microprocessors Using Explainable ML". In: 22nd International Workshop on Worst-Case Execution Time Analysis (WCET 2024). Ed. by Thomas Carle. Vol. 121.
[78] Simon Wegener et al. "EnergyAnalyzer: Using Static WCET Analysis Techniques to Estimate the Energy Consumption of Embedded Applications". en. In: Schloss Dagstuhl – Leibniz-Zentrum f¨ur Informatik, 2023. doi: 10.4230/OASICS.WCET.2023.9. url: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2023.9.

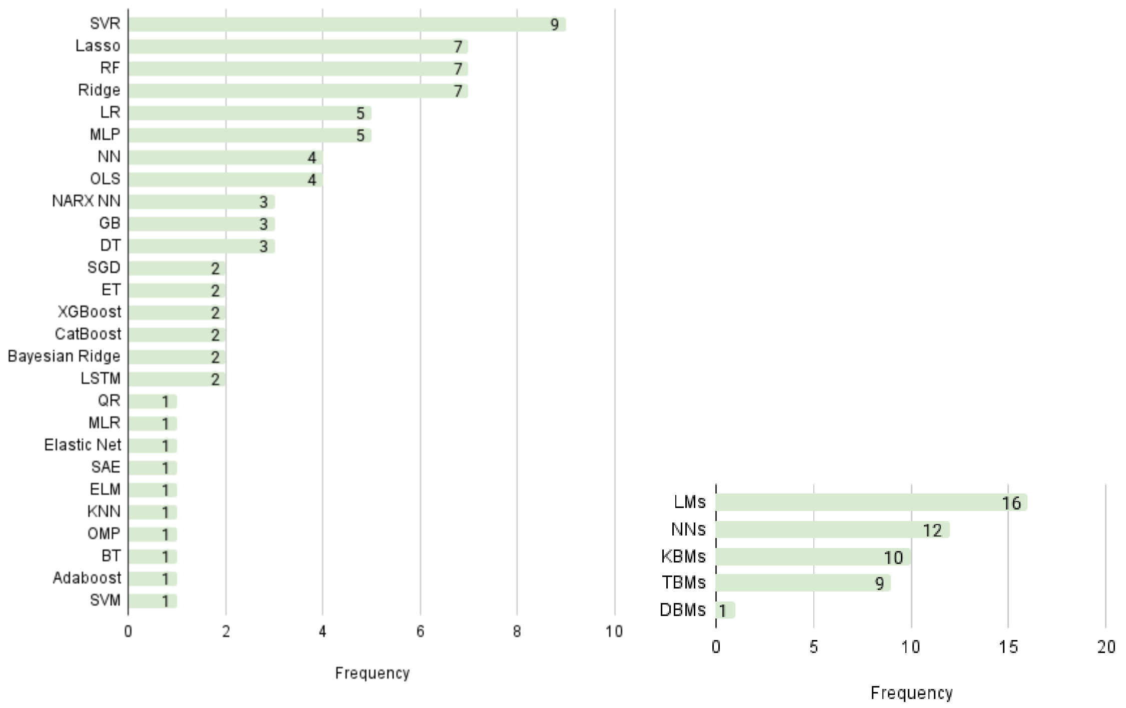


Figure 6. Models used for predicting energy consumption

More advanced approaches focus on capturing nonlinear dependencies through ensemble and neural models. Ensemble methods such as Random Forest, Gradient Boosting, XGBoost, CatBoost, Extra Trees, and Bagging consistently appear as options across several studies[79, 80]. Neural network models are also extensively employed, ranging from Multi-Layer Perceptrons and Long Short-Term Memory (LSTM) networks to domain-specific architectures like NARX Neural Nets[81, 82, 83] and Stacked Auto-Encoders[84]. These models provide greater expressive power, enabling the modelling of complex interactions between the features and the power or EC, though at the cost of interpretability.

[79] Gargi Alavani Prabhu et al. “Estimating Power Consumption of GPU Application Using Machine Learning Tool”. In: 2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI). Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2024, pp. 734–739. doi: 10.1109/ICTAI62512. 2024.00109. url: https://doi.ieeecomputersociety.org/10.1109/ICTAI62512.2024.00109.

[80] Gargi Alavani et al. “Program Analysis and Machine Learning–based Approach to Predict Power Consumption of CUDA Kernel”. In: ACM Trans. Model. Perform. Eval. Comput. Syst. 8.4 (July 2023). issn: 2376-3639. doi: 10.1145/3603533. url: https://doi.org/10.1145/3603533.

[81] Oussama Djedidi and Mohand Djeziri. “Power profiling and monitoring in embedded systems: A comparative study and a novel methodology based on NARX neural networks”. In: Journal of Systems Architecture 111 (Dec. 2020), p. 101805. doi: 10.1016/j.sysarc.2020.101805. url: https://amu.hal.science/hal-02740661.

[82] Oussama Djedidi et al. “A Novel Easy-to-construct Power Model for Embedded and Mobile Systems”. In: 15th International Conference on Informatics in Control, Automation and Robotics. SCITEPRESSScience and Technology Publications. 2018.

[83] Oussama Djedidi et al. “Constructing an accurate and a high-performance power profiler for embedded systems and smartphones”. In: Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems. 2018, pp. 79–82.

[84] Muhammed Maruf Ozturk. “Tuning stacked auto-encoders for energy consumption prediction: a case study”. In: International Journal of Information Technology and Computer Science 11.2 (2019), pp. 1–8.

Some works leverage hybrid strategies and combine linear models with neural networks or tree-based models to balance interpretability and prediction accuracy[85]. Others just rely on classical models[86].

The left chart in Figure 7 shows the frequency of each individual model. The most frequently used are Support Vector Regression (SVR) models (9 Studies), followed by Lasso Regression, Random Forest, and Ridge Regression (7 Studies). Multi-Layer Perceptron and Linear Regression were used in 5 studies, whereas Neural Networks and Ordinary Least Squares in 4 studies each. The remaining models appear less frequently.

The chart on the right in Figure 7 presents the same models, we grouped into broader categories. Linear models are the most represented cluster (16 studies), followed by neural networks (12 studies), kernel-based models (10 studies), decision-tree models (9 studies), and distance-based models (1 study). This indicates a strong reliance on linear models and neural-networks, complemented by kernel-based and ensemble approaches for modelling software energy consumption. Most of the studies analysed evaluate and compare multiple models; Studies that focus on one model include Multi-Layer Perceptron[87], Random Forest[88], NARX Neural Network, Ordinary Least Squares [89, 90, 91, 92], Stacked Auto-Encoders[93], Extreme Learning Machine[94], Support Vector Regression[95], and Lasso Regression[96].

---

[85] Tong Zhang et al. “Assessing Predictive Models for Energy Consumption Across Varied Software Environments”. In: 2024 IEEE International Conference on Big Data (BigData). Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2024, pp. 5233–5242. doi: 10.1109/BigData62323.2024.10825500. url: https://doi.ieeecomputersociety.org/10.1109/BigData62323.2024.10825500.

[86] Simon Wegener et al. “EnergyAnalyzer: Using Static WCET Analysis Techniques to Estimate the Energy Consumption of Embedded Applications”. en. In: Schloss Dagstuhl – Leibniz-Zentrum f¨ur Informatik, 2023. doi: 10.4230/OASICS.WCET.2023.9. url: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2023.9.

[87] Kris Nikov et al. “Accurate Energy Modelling on the Cortex-M0 Processor for Profiling and Static Analysis”.
In: 2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS). IEEE, Oct. 2022, 1–4. doi: 10.1109/icecs202256217.2022.9971086. url: http://dx.doi.org/10.1109/ICECS202256217.2022.9971086.

[88] Lorenz Braun et al. A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels. 2020. arXiv: 2001.07104 [cs.DC]. url: https://arxiv.org/abs/2001.07104.

[89] Simon Wegener et al. “EnergyAnalyzer: Using Static WCET Analysis Techniques to Estimate the Energy Consumption of Embedded Applications”. en. In: Schloss Dagstuhl – Leibniz-Zentrum f¨ur Informatik, 2023. doi: 10.4230/OASICS.WCET.2023.9. url: https://drops.dagstuhl.de/entities/document/10. 4230/OASIcs.WCET.2023.9.

[90] Kris Nikov et al. “Robust and Accurate Fine-Grain Power Models for Embedded Systems with No On-Chip PMU”. In: IEEE Embedded Systems Letters 14.3 (Sept. 2022), 147–150. issn: 1943-0671. doi: 10.1109/les.2022.3147308. url: http://dx.doi.org/10.1109/LES.2022.3147308.

[91] Krastin Nikov and Jose Nunez-Yanez. “Intra and inter-core power modelling for single-ISA heterogeneous
processors”. In: International Journal of Embedded Systems 12 (Jan. 2020), p. 324. doi: 10.1504/IJES.2020.107046.

[92] Matthew J Walker et al. “Accurate and stable run-time power modeling for mobile and embedded CPUs”. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 36.1 (2016), pp. 106–119.

[93] Muhammed Maruf Ozturk. “Tuning stacked auto-encoders for energy consumption prediction: a case study”. In: International Journal of Information Technology and Computer Science 11.2 (2019), pp. 1–8.

[94] Deguang Li et al. “Software Energy Consumption Estimation at Architecture-Level”. In: 2016 13th International Conference on Embedded Software and Systems (ICESS). IEEE. 2016, pp. 7–11.

[95] Xiong Wei et al. “An embedded software power consumption model based on software architecture and support vector machine regression”. In: International Journal of Smart Home 10.3 (2016), pp. 191–200.

[96] Xinnian Zheng, Lizy K John, and Andreas Gerstlauer. “Accurate phase-level cross-platform power and performance estimation”. In: Proceedings of the 53rd Annual Design Automation Conference. 2016, pp. 1–6.

We additionally categorized the EC models with respect to the type of their input, where we followed the classification by Hoque et al.[97]. The left chart in Figure 7 also shows the frequency distribution of the different model types. Event-based models are the most prevalent, appearing in 11 studies, which indicates a strong focus on capturing runtime system behavior through events such as system calls. Code-analysis-based models follow with 6 occurrences, reflecting continued interest in static estimation approaches that do not require executing the software. Utilization-based models appear in 5 studies, showing moderate adoption for correlating resource usage with power consumption. Finally, 4 studies employ hybrid approaches combining multiple input sources, for instance utilization-based with event-based approaches.

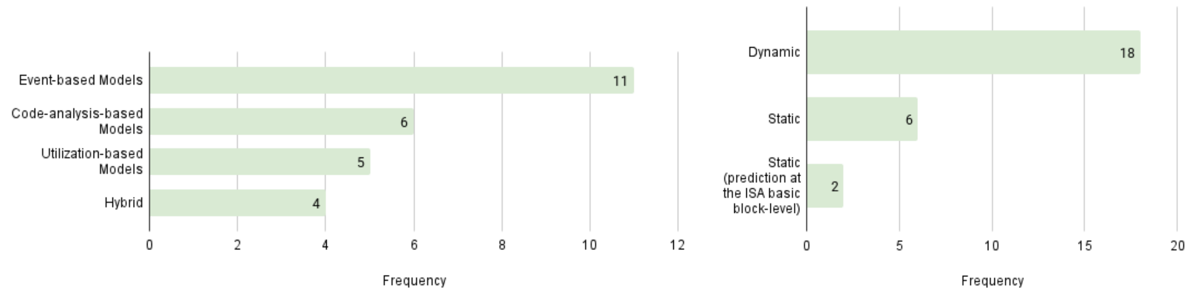


Figure 7. Model types regarding the input for prediction

Furthermore, we classified the approaches according to the predicting method uses distinguished static or dynamic methods. Static prediction methods do not require executing software to predict its energy consumption, whereas dynamic methods do require running the software. Some methods support both static and dynamic prediction, depending on the type of input they are provided with. We classify these methods as static. The right chart of Figure 7 shows that most reviewed studies (18) follow a dynamic approach. The 6 studies follow a static approach, with one study[98], in which the input to the prediction method includes, in addition to software code, a runtime information regarding loops, e.g., the number of loop iterations. Because this additional information may be approximated by the model's user without running the code, we classified it as static one. If this input cannot be approximated, the code must be executed and the prediction approach becomes a dynamic one. We classified the remaining two models as multimodal because EC predictions can be done either with static or dynamic inputs. Each of these models consists of two sub-models arranged in a pipeline, in which the output of the first sub-model is the input to the second one. In both cases, the first sub-model estimates PMCs at the Instruction Set architecture (ISA) basic block level based on software code, while the second takes the estimated PMCs to predict the energy consumed. In this sense, the first sub-model in the pipeline is a static one whereas the second a dynamic one. The model can be provided with a software code or with PMCs. Regarding the first sub-model, that predicts PMCs basedon code, Wegener et. al. [99] use microarchitectural analysis to predict an upper bound of the various PMCs for each instruction in

[97] Mohammad Ashraful Hoque et al. "Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices". In: ACM Comput. Surv. 48.3 (Dec. 2015). issn: 0360-0300. doi: 10.1145/2840723. url: https://doi.org/10.1145/2840723.

[98] Charalampos Marantos et al. "A Flexible Tool for Estimating Applications Performance and Energy Consumption Through Static Analysis". In: SN Comput. Sci. 2.1 (Jan. 2021). doi: 10.1007/s42979-020-00405-7. url: https://doi.org/10.1007/s42979-020-00405-7.

[99] Simon Wegener et al. "EnergyAnalyzer: Using Static WCET Analysis Techniques to Estimate the Energy Consumption of Embedded Applications". en. In: Schloss Dagstuhl – Leibniz-Zentrum f¨ur Informatik, 2023. doi: 10.4230/OASICS.W

the Control Flow Diagram (CFG). Nikov et. al. [100] predicts PMCs with architecture models. Unfortunately, neither of the two studies provides detailed specification of these sub-models. In both cases, training the first, static, sub-model requires executing software code and collecting actual PMCs for the predefined code.

- RQ5: How and with what tools is the actual energy consumption measured?

Two main approaches to measuring energy consumption are hardware-based and software-based (Figure 8). Hardware-based methods rely on integrated power monitoring units (PMUs) or external sensors. These methods provide high-accuracy measurements but are more complex and expensive to set up than software solutions, so they are less commonly used. By contrast, software-based approaches use system profilers and APIs to estimate energy consumption and are the most popular (12 studies). Common tools include *perf*[101], NVIDIA Management Library (NVML)[102, 103], Intel Power Gadget[104], and CUDA Flux[105]. Hybrid frameworks such as GreenMiner[106, 107, 108] combine software logging with hardware instrumentation to enable large-scale energy data collection on Android devices. Although software methods offer greater flexibility and broader applicability, they typically achieve lower accuracy than direct hardware measurements and are more sensitive to system-level noise. Simulation-based tools, such as MAGEEC[109] and HMSim[110], also appear in the literature but are used less frequently.

[100] Kris Nikov et al. "Accurate Energy Modelling on the Cortex-M0 Processor for Profiling and Static Analysis".
In: 2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS). IEEE, Oct. 2022, 1–4. doi: 10.1109/icecs202256217.2022.9971086. url: http://dx.doi.org/10.1109/ICECS202256217.2022.9971086.
[101] Tong Zhang et al. "Assessing Predictive Models for Energy Consumption Across Varied Software Environments". In: 2024 IEEE International Conference on Big Data (BigData). Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2024, pp. 5233–5242. doi: 10.1109/BigData62323.2024.10825500. url: https://doi.ieeecomputersociety.org/10.1109/BigData62323.2024.10825500.
[102] Gargi Alavani Prabhu et al. "Estimating Power Consumption of GPU Application Using Machine Learning Tool". In: 2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI).
Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2024, pp. 734–739. doi: 10.1109/ICTAI62512.2024.00109. url: https://doi.ieeecomputersociety.org/10.1109/ICTAI62512.2024.00109.
[103] Gargi Alavani et al. "Program Analysis and Machine Learning–based Approach to Predict Power Consumption
of CUDA Kernel". In: ACM Trans. Model. Perform. Eval. Comput. Syst. 8.4 (July 2023). issn: 2376-3639. doi: 10.1145/3603533. url: https://doi.org/10.1145/3603533.
[104] Muhammed Maruf ¨ Ozt¨urk. "Tuning stacked auto-encoders for energy consumption prediction: a case study". In: International Journal of Information Technology and Computer Science 11.2 (2019), pp. 1–8.
[105] Lorenz Braun et al. A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels. 2020. arXiv: 2001.07104 [cs.DC]. url: https://arxiv.org/abs/2001.07104.
[106] Shaiful Alam Chowdhury and Abram Hindle. "Greenoracle: Estimating software energy consumption with energy measurement corpora". In: Proceedings of the 13th international conference on mining software repositories. 2016, pp. 49–60.
[107] Shaiful Alam Chowdhury et al. "A system-call based model of software energy consumption without hardware instrumentation". In: 2015 Sixth International Green and Sustainable Computing Conference (IGSC). IEEE. 2015, pp. 1–6.
[108] Stephen Romansky et al. "Deep green: Modelling time-series of software energy consumption". In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE. 2017, pp. 273–283.
[109] Shivam Kundan, Ourania Spantidi, and Iraklis Anagnostopoulos. "Online frequency-based performance and power estimation for clustered multi-processor systems". In: Computers Electrical Engineering 90 (Mar. 2021), p. 106971. doi: 10.1016/j.compeleceng.2021.106971.
[110] Xiong Wei et al. "An embedded software power consumption model based on software architecture and support vector machine regression". In: International Journal of Smart Home 10.3 (2016), pp. 191–200.

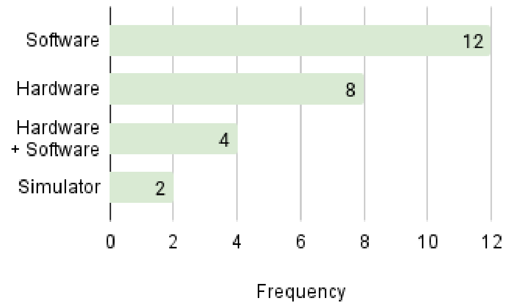


Figure 8. Methods and tools for measuring actual energy consumption

- RQ6: What environment is the software operating in and how it is configured?

The studies analysed measure the energy consumption of software on a wide variety of hardware platforms, which can be grouped into several categories. Several works use traditional server-class machines such as Linux servers[111, 112]nd specialized infrastructures, for example the Marcher2 server[113]. GPU-based systems are widely represented, with accelerators such as NVIDIA Tesla K20, K80, M60, V100, and more recent GPUs such as RTX A4000, RTX 4060, GTX1650, Titan Xp, and P100[114, 115, 116]. These configurations are mostly used for CUDA-based workloads. On the embedded side, GPU-enabled boards such as the NVIDIA Jetson TX1 and Jetson Xavier NX are employed to capture energy behavior in constrained environments[117, 118]. Embedded boards and microcontrollers are another frequent target. Examples include ARM Cortex-M0 (STM32F0-

---

[111] Deguang Li et al. “Software Energy Consumption Estimation at Architecture-Level”. In: 2016 13th International Conference on Embedded Software and Systems (ICESS). IEEE. 2016, pp. 7–11.

[112] Tong Zhang et al. “ Assessing Predictive Models for Energy Consumption Across Varied Software Environments”. In: 2024 IEEE International Conference on Big Data (BigData). Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2024, pp. 5233–5242. doi: 10.1109/BigData62323.2024.10825500. url: https://doi.ieeecomputersociety.org/10.1109/BigData62323.2024.10825500.

[113] Cuijiao Fu, Depei Qian, and Zhongzhi Luan. “Estimating software energy consumption with machine learning approach by software performance feature”. In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). IEEE. 2018, pp. 490–496.

[114] Gargi Alavani Prabhu et al. “ Estimating Power Consumption of GPU Application Using Machine Learning Tool ”. In: 2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI). Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2024, pp. 734–739. doi: 10.1109/ICTAI62512. 2024.00109. url: https://doi.ieeecomputersociety.org/10.1109/ICTAI62512.2024.00109.

[115] Gargi Alavani et al. “Program Analysis and Machine Learning–based Approach to Predict Power Consumption of CUDA Kernel”. In: ACM Trans. Model. Perform. Eval. Comput. Syst. 8.4 (July 2023). issn: 2376-3639. doi: 10.1145/3603533. url: https://doi.org/10.1145/3603533.

[116] Lorenz Braun et al. A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels. 2020. arXiv: 2001.07104 [cs.DC]. url: https://arxiv.org/abs/2001.07104.

[117] Charalampos Marantos, Nikolaos Maidonis, and Dimitrios Soudris. “Designing Application Analysis Tools for Cross-Device Energy Consumption Estimation”. In: 2022 11th International Conference on Modern Circuits and Systems Technologies (MOCAST). 2022, pp. 1–4. doi: 10.1109/MOCAST54814.2022.9837632.

[118] Charalampos Marantos et al. “A Flexible Tool for Estimating Applications Performance and Energy Consumption Through Static Analysis”. In: SN Comput. Sci. 2.1 (Jan. 2021). doi: 10.1007/s42979-020-00405-7. url: https://doi.org/10.1007/s42979-020-00405-7.

Discovery board)[119], [120], MSP430FR5969[121], single-core ARM7[122], and LEON3 GR712RC processors, sometimes combined with FPGAs like Kintex UltraScale for hybrid experiments)[123, 124]. ARM-based development boards, such as Odroid-U3 and Odroid-XU3, which embed Cortex-A15 and Cortex-A7 clusters, are also widely used for heterogeneous embedded scenarios[125, 126, 127, 128]. Finally, mobile and consumer devices are prominent in many studies. Android smartphones constitute a significant portion of the evaluation platforms, often combined with external measurement hardware such as GreenMiner for improved accuracy[129, 130, 131, 132].

Figure 9 summarizes the frequency of system types across the surveyed studies. Microcontrollers (MCUs) are the most frequently evaluated platforms (5 studies) followed by Android smartphones,

---

[119] Simon Wegener et al. "EnergyAnalyzer: Using Static WCET Analysis Techniques to Estimate the Energy Consumption of Embedded Applications". en. In: Schloss Dagstuhl – Leibniz-Zentrum f¨ur Informatik, 2023. doi: 10.4230/OASICS.WCET.2023.9. url: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2023.9.

[120] Kris Nikov et al. "Accurate Energy Modelling on the Cortex-M0 Processor for Profiling and Static Analysis".
In: 2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS). IEEE, Oct. 2022, 1–4. doi: 10.1109/icecs202256217.2022.9971086. url: http://dx.doi.org/10.1109/ICECS202256217.2022.9971086.

[121] Hugo Reymond, Abderaouf Nassim Amalou, and Isabelle Puaut. "WORTEX: Worst-Case Execution Time and Energy Estimation in Low-Power Microprocessors Using Explainable ML". In: 22nd International Workshop on Worst-Case Execution Time Analysis (WCET 2024). Ed. by Thomas Carle. Vol. 121. Open Access Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 1:1–1:14. isbn: 978-3-95977-346-1. doi: 10.4230/OASIcs.WCET.2024.1. url: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2024.1.

[122] Xiong Wei et al. "An embedded software power consumption model based on software architecture and support vector machine regression". In: International Journal of Smart Home 10.3 (2016), pp. 191–200.

[123] Simon Wegener et al. "EnergyAnalyzer: Using Static WCET Analysis Techniques to Estimate the Energy Consumption of Embedded Applications". en. In: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi: 10.4230/OASICS.WCET.2023.9. url: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2023.9.

[124] Kris Nikov et al. "Accurate Energy Modelling on the Cortex-M0 Processor for Profiling and Static Analysis".
In: 2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS). IEEE, Oct. 2022, 1–4. doi: 10.1109/icecs202256217.2022.9971086. url: http://dx.doi.org/10.1109/ICECS202256217.2022.9971086.

[125] Matthew J Walker et al. "Accurate and stable run-time power modeling for mobile and embedded CPUs". In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 36.1 (2016), pp. 106–119.

[126] Xinnian Zheng, Lizy K John, and Andreas Gerstlauer. "Accurate phase-level cross-platform power and performance estimation". In: Proceedings of the 53rd Annual Design Automation Conference. 2016, pp. 1–6.

[127] Krastin Nikov and Jose Nunez-Yanez. "Intra and inter-core power modelling for single-ISA heterogeneous
processors". In: International Journal of Embedded Systems 12 (Jan. 2020), p. 324. doi: 10.1504/IJES.2020.107046.

[128] Shivam Kundan, Ourania Spantidi, and Iraklis Anagnostopoulos. "Online frequency-based performance and power estimation for clustered multi-processor systems". In: Computers Electrical Engineering 90 (Mar. 2021), p. 106971. doi: 10.1016/j.compeleceng.2021.106971.

[129] Shaiful Chowdhury et al. "Greenscaler: training software energy models with automatic test generation". In: Empirical Software Engineering 24.4 (2019), pp. 1649–1692.

[130] Shaiful Alam Chowdhury and Abram Hindle. "Greenoracle: Estimating software energy consumption with energy measurement corpora". In: Proceedings of the 13th international conference on mining software repositories. 2016, pp. 49–60.

[131] Shaiful Alam Chowdhury et al. "A system-call based model of software energy consumption without hardware instrumentation". In: 2015 Sixth International Green and Sustainable Computing Conference (IGSC). IEEE. 2015, pp. 1–6.

[132] Stephen Romansky et al. "Deep green: Modelling time-series of software energy consumption". In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE. 2017, pp. 273–283.

either standalone or paired with Green Miner hardware (4 studies), reflecting the importance of energy measurement for mobile devices. Embedded ARM clusters also appear in 4 studies. GPU-equipped systems, including NVIDIA GPUs, are present in 3 studies, while traditional server-class machines, such as Linux servers, and NVIDIA embedded boards appear in 2 studies each. A few studies use less common or unspecified platforms: one study does not specify the system used, and two studies are grouped under "Other", which includes Raspberry Pi 4, Intel i5-4210U, and the Marcher2 server.

Overall, this distribution highlights a strong focus on microcontrollers, embedded ARM clusters, and mobile devices, complemented by GPUs and server-class machines.

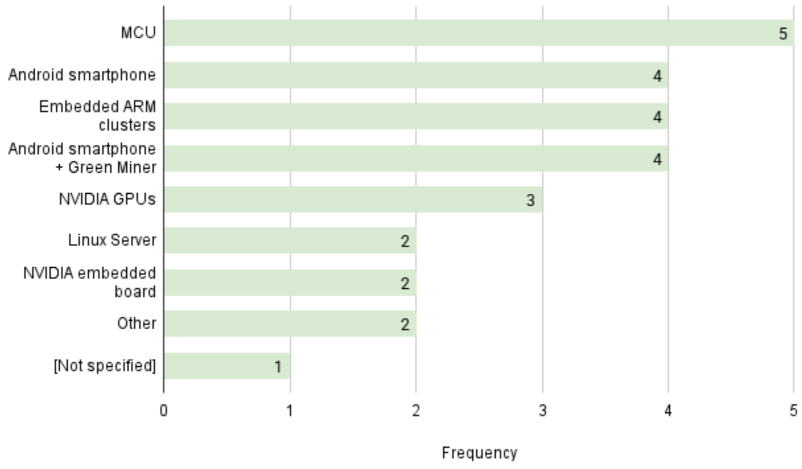


Figure 9. Energy prediction environments and settings

➢ Limitations & gaps

The analysis of related work indicated a couple of relevant research gaps regarding the prediction of software energy consumption (Table 4). The most significant gap is the lack of approaches addressing Python programming language despite the rapid increase of this usage, especially in the area of data analysis (data science, machine learning and artificial intelligence) and development of data-driven software systems. Further research deficit relates to a static code analysis, specifically measuring static properties of code at fine-grained level for the purpose of predicting energy consumption.

Table 4. Gaps regarding the prediction of software energy consumption

Limitation/Gap	Description	Example/Implication	Application To GreenCode
Limited support for Python language	Very scarce research on predicting energy consumption of Python software based on source code analysis	Data analysis and data-driven software applications written in Python cannot be analysed and optimized concerning energy consumption.	One of the focus areas of GreenCode are energy intensive data processing software systems.
Limited support for fine-grained code analysis	Missing method for analysing source code for fine-grained energy consumption predictions.	Individual code structures that are responsible for high energy consumption cannot be spotted because only large junks of software are considered.	GreenCode aims at optimizing energy consumption. This requires identifying exact location in energy-intensive parts of software code, which may be on various granularity levels, including fine-grained ones.

4.3 Benchmark Analysis

This section presents a structured review of existing benchmarks that evaluate Large Language Models (LLMs) in the context of code generation and optimization, with an emphasis on efficiency and sustainability. The analysis is guided by five research questions (RQs), each examining a specific dimension of benchmark design, evaluation methodology, LLM interaction, and observed limitations. The corresponding findings are consolidated in Table 5 and Table 6, which provide comparative overviews of the benchmark landscape and evaluation metrics.

- RQ1: Which code optimization benchmarks exist?
- RQ2: Which programming languages do they cover?
- RQ3: What types of data do they use (function-level, codebase-level, synthetic example)?
- RQ4: What criteria are used to evaluate optimizations (runtime, memory, energy, etc.)?
- RQ5: What are the main limitations of current benchmarks?

The identified benchmarks exhibit variation in their design scope, programming language coverage, and data granularity. To facilitate comparison, Table 5 provides an overview of representative benchmarks. The Language column indicates the primary implementation languages used for benchmark tasks, while the Data Scope column specifies the granularity level ranging from function-level and program-level examples to repository-scale datasets. The Description column briefly summarizes each benchmark's unique objective or methodological focus.

Function-level evaluations dominate the current landscape, largely due to their simplicity and high reproducibility. Python remains the predominant language, reflecting its centrality in LLM-based code generation research. Program-level and repository-level evaluations are relatively rare, signalling a gap in real-world efficiency assessments that consider cross-module dependencies and runtime environments.

Table 5. Benchmark Landscape, Language Coverage, and Data Types

Benchmark	Year	Language(s)	Data Scope	Description
PCEBench[133]	2025	C/C++	Function/Program	Parallel code generation using OpenMP/MPI
ResBench[134]	2025	Verilog	Function	FPGA design generation with resource constraints
MARCO[135]	2025	Python	Program/HPC	Multi-agent optimization for HPC kernels
EffiBench-X[136]	2025	Python, C++, Java, JS, Ruby, Go	Function	Multi-language benchmark for runtime and memory efficiency

[133] L. Chen, N. Ahmed, M. Capotă, T. Willke, N. Hasabnis and A. Jannesari, "PCEBench: A Multi-Dimensional Benchmark for Evaluating Large Language Models in Parallel Code Generation," 2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Milano, Italy, 2025, pp. 546-557, doi: 10.1109/IPDPS64566.2025.00055. keywords: {Technological innovation;Codes;Parallel programming;Large language models;Scalability;Benchmark testing;Software systems;Multitasking;Natural language processing;Synchronization;large language model;parallel code generation;benchmark;evaluation;LLM agent},

[134] Hu, P., Pan, W., Jian, X., Ma, Z., Li, T., Shen, Y., ... & Li, Z. (2025). ResBench: A Comprehensive Framework for Evaluating Database Resilience. *arXiv preprint arXiv:2511.11088*.

[135] Asif Rahman, Veljko Cvetkovic, Kathleen Reece, Aidan Walters, Yasir Hassan, Aneesh Tummeti, Bryan Torres, Denise Cooney, Margaret Ellis, and Dimitrios S. Nikolopoulos. [n. d.]. Performance Evaluation of Large Language Models for High-Performance Code Generation: A Multi-Agent Approach (MARCO). https://api.semanticscholar.org/CorpusID: 280547604

[136] Qing, Y., Zhu, B., Du, M., Guo, Z., Zhuo, T. Y., Zhang, Q., ... & Tuan, L. A. (2025). EffiBench-X: A Multi-Language Benchmark for Measuring Efficiency of LLM-Generated Code. *arXiv preprint arXiv:2505.13004*.

ENAMEL[137]	2024	Python	Function	Function-level efficiency benchmark based on HumanEval/MBPP
Coffe[138]	2025	Python	Function/Program	CPU instruction-based efficiency benchmark
ECCO[139]	2024	Python	Function	Natural-language instructed code optimization
CodeEditorBench[140]	2024	C++, Java, Python	Function	Code editing and performance improvement tasks
Mercury[141]	2024	Python	Function	Runtime-weighted efficiency assessment
EffiBench[142]	2024	Python	Function	Efficiency measurement via runtime and memory profiling
EvalPerf[143]	2024	Python	Function	Hardware counter-based differential performance evaluation
SWE-Perf[144]	2025	Python	Repository	Real-world pull request–based optimization evaluation
RACE[145]	2024	Python	Function	Benchmark evaluating multi-dimensional code generation beyond correctness; measures time and space complexity, readability, and maintainability.

- RQ4: What criteria are used to evaluate optimizations (runtime, memory, energy, etc.)?

Evaluation criteria in current benchmarks predominantly focus on runtime and memory efficiency, reflecting the traditional emphasis on computational speed in software performance analysis. Nonetheless, several recent studies broaden this perspective by introducing multi-dimensional efficiency metrics that account for hardware-level performance, normalized composite indicators, and instruction-based stability measures.

Table 6 presents an overview of benchmarks that include quantitative evaluations of efficiency, illustrating the variety of metrics and methodologies employed to assess the performance of LLM-generated or optimized code.

[137] Qiu, R., Zeng, W. W., Ezick, J., Lott, C., & Tong, H. (2024). How efficient is llm-generated code? a rigorous & high-standard benchmark. *arXiv preprint arXiv:2406.06647*.
[138] Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. 2025. COFFE: A Code Efficiency Benchmark for Code Generation. arXiv:2502.02827 [cs.SE] https://arxiv.org/abs/2502.02827
[139] Siddhant Waghjale, Vishruth Veerendranath, Zora Zhiruo Wang, and Daniel Fried. 2024. ECCO: Can We Improve Model-Generated Code Efficiency Without Sacrificing Functional Correctness? arXiv:2407.14044 [cs.CL] https://arxiv.org/abs/2407.14044
[140] Guo, J., Li, Z., Liu, X., Ma, K., Zheng, T., Yu, Z., ... & Fu, J. (2024). Codeeditorbench: Evaluating code editing capability of large language models. *arXiv preprint arXiv:2404.03543*.
[141] Mingzhe Du, Luu Anh Tuan, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: a code efficiency benchmark for code large language models. In Proceedings of the 38th International Conference on Neural Information Processing Systems (Vancouver, BC, Canada) (NIPS '24). Curran Associates Inc., Red Hook, NY, USA, Article 529, 22 pages.
[142] Dong HUANG, Yuhao QING, Weiyi Shang, Heming Cui, and Jie Zhang. 2024. EffiBench: Benchmarking the Efficiency of Automatically Generated Code. In The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track. https://openreview.net/forum?id=30XanJanJP
[143] Liu, J., Xie, S., Wang, J., Wei, Y., Ding, Y., & Zhang, L. (2024). Evaluating language models for efficient code generation. *arXiv preprint arXiv:2408.06450*.
[144] He, X., Liu, Q., Du, M., Yan, L., Fan, Z., Huang, Y., ... & Ma, Z. (2025). Swe-perf: Can language models optimize code performance on real-world repositories?. *arXiv preprint arXiv:2507.12415*.
[145] Zheng, J., Cao, B., Ma, Z., Pan, R., Lin, H., Lu, Y., ... & Sun, L. (2024). Beyond correctness: Benchmarking multi-dimensional code generation for large language models. *arXiv preprint arXiv:2407.11470*.

Table 6. Evaluation Criteria Across Benchmarks

Benchmark	Metrics Used
PCEBench	Execution time, RaceFree@k, Correct@k, SpeedPass@k (speedup ≥1.5×)
ResBench	LUT usage, synthesis success, functional correctness
MARCO	Execution time, FLOPS, memory usage, readability, cost efficiency
EffiBench-X	Execution Time, Memory Peak, Memory Integral, Pass@1
ENAMEL	Efficiency@k (eff@1), correctness, runtime stability
RACE	Time complexity, space complexity, correctness, readability, maintainability
Coffe	Efficient@k, CPU instruction count, speedup ratio, Pass@k
ECCO	Runtime speedup, memory reduction, correctness, % optimized
CodeEditorBench	Pass@1, Mean OptScore, readability
Mercury	Beyond metric (runtime-weighted Pass), correctness
EffiBench	Execution Time, Max Memory Usage, Total Memory Usage, and their normalized versions (NET, NMU, NTMU)
EvalPerf	DPS_norm (normalized performance score), correctness
SWE-Perf	Runtime gain, performance ratio, correctness (test suite)

As summarized in Table 6, existing benchmarks collectively span a broad range of performance and quality metrics yet remain heterogeneous in methodology and granularity. Most studies such as EffiBench-X, EffiBench, and ENAMEL focus primarily on runtime and memory efficiency, offering fine-grained yet narrowly scoped evaluations. In contrast, RACE and CodeEditorBench incorporate qualitative measures like readability, maintainability, and complexity compliance, broadening the evaluation scope toward developer-centric metrics.

Hardware-oriented benchmarks such as EvalPerf, Coffe, and ResBench, introduce more reproducible, platform-aware measures, including hardware counters, instruction counts, and resource utilization. Meanwhile, SWE-Perf and MARCO represent the highest level of realism by benchmarking within complete execution environments (Dockerized repositories or multi-agent HPC systems).

Despite these advances, energy consumption and carbon impact remain absent as direct measurements. Current approaches rely on surrogate metrics like runtime, instruction count, or hardware counters that provide only partial approximations of true energy efficiency. Establishing standardized, energy-calibrated protocols therefore remains a key objective for the GreenCode benchmark suite, ensuring fair, reproducible, and sustainability-aware evaluation of LLM-driven code optimization.

- RQ5: What are the main limitations of current benchmarks?

Despite major advancements in LLM-based code benchmarking, several critical limitations continue to constrain their accuracy, comparability, and practical applicability. These limitations are methodological, technical, and conceptual, highlighting the need for more comprehensive and sustainability-aware evaluation frameworks.

- Limited Granularity and Realism: Most existing benchmarks focus on isolated, function-level tasks, which are suitable for controlled experiments but fail to capture the complexity of real-world software systems. The absence of large-scale, program-level or repository-level evaluations restricts understanding of end-to-end optimization behaviour.
- Lack of Energy and Sustainability Metrics: Current benchmarks measure performance primarily through runtime or memory usage. However, they do not include direct

measurements of energy consumption, power usage, or carbon impact. This omission prevents a full assessment of sustainability-related trade-offs.
- Data Contamination and Benchmark Saturation: Many benchmarks rely on widely used public datasets, which are often part of model training corpora. This overlap introduces bias, overestimates model performance, and undermines the validity of comparative evaluations.
- Inconsistent Experimental Environments: Benchmarks are typically executed on heterogeneous hardware setups without standardized conditions or containerized environments. As a result, reproducibility across studies remains limited, and direct comparison of results is unreliable.
- Narrow Evaluation Objectives: Most benchmarks emphasize correctness and runtime improvement while overlooking complementary software qualities such as maintainability, readability, or robustness. A sustainable benchmark must integrate these dimensions to better represent real-world development goals.

Overall, existing benchmarking methodologies remain fragmented and primarily performance driven. Their lack of environmental and contextual awareness limits their contribution to sustainable software engineering. Establishing standardized, reproducible benchmarks is therefore a necessary step toward evaluating LLM-generated code in realistic and sustainability-oriented contexts.

4.4 Quality assessment of genAI outcomes

The evaluation of generative AI (GenAI) outputs has become a critical research focus across domains such as computer science, healthcare, and law. Recent studies emphasise that *quality* is a complex, multidimensional, construct that comprises factual accuracy, completeness, reasoning coherence, clarity and style, safety, and trustworthiness [146, 147]. In certain contexts, these criteria extend beyond linguistics to include domain-specific notions to position GenAI as a component for decision[148].

Traditional metrics such as BLEU and ROUGE have proven inadequate for open-ended or creative GenAI tasks because they reward surface similarity rather than faithfulness or usefulness[149]. Consequently, *LLM-as-a-judge* approaches, in which a strong model evaluates another model's output, have gained traction[150]. Frameworks like G-Eval demonstrate closer alignment with human judgements on factuality and coherence than older metrics, though evaluator bias and self-agreement remain open concerns[151].

In software engineering contexts, researchers increasingly examine GenAI-generated artefacts beyond fluency and task accuracy, measuring code maintainability, security vulnerabilities and

[146] Tam, T.Y.C., et al. (2024) 'A framework for human evaluation of large language models in healthcare derived from literature review', *npj Digital Medicine / arXiv:2405.02559*.
[147] Budler, L.C. et al. (2025) 'A brief review on benchmarking for large language models', *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*
[148] Asgari, E. et al. (2025) 'A framework to assess clinical safety and hallucination risk in large language model outputs', *npj Digital Medicine*.
[149] Liu, Y. et al. (2023) 'G-Eval: NLG evaluation using GPT-4 with better human alignment', *Proceedings of EMNLP 2023*.
[150] Wang, Y. et al. (2025) 'LLM-as-a-Judge: Reliability, bias, and best practices', *arXiv preprint*.
[151] Liu, Y. et al. (2023) 'G-Eval: NLG evaluation using GPT-4 with better human alignment', *Proceedings of EMNLP 2023*

static-analysis defects[152] [153] [154]. In the automotive domain, where software must satisfy real-world hardware constraints and strict safety standards, recent studies propose frameworks that integrate GenAI with formal verification, requirement-analysis pipelines and system-level validation to enable trustworthy generation[155] [156] [157].

Other recent studies in software engineering and embedded systems extend quality assessment of generative AI into domain-specific artefacts such as code generation and safety-critical embedded software. For instance, the CODEJUDGE framework proposes using a large language model to evaluate the *semantic correctness* of generated code without relying solely on test cases, thereby addressing limitations of purely functional metrics[158]. Empirical work examining LLM-generated code goes further: one study evaluated maintainability and reliability of Python code produced by different model configurations (zero-/few-shot, fine-tuned) using static analysis and found weak correlation between functional correctness and code quality issues like code smells and defects[159]. Another line of research addresses safety-critical domains: e.g., studies in automotive software ask whether GenAI-generated code can comply with verification and simulation constraints inherent to the automotive software development lifecycle [160] [161]. Benchmarks such as L2CEval further analyse language-to-code generation capability across tasks and emphasise calibration and error-analysis in code generation quality, underscoring that correctness is only one axis of quality[162]. These domain-specific advances underline that for code-generation, “quality” must be judged not just by whether it runs, but by maintainability, security-vulnerability risk, integration correctness, and compliance with domain verification.

Another key focus of recent research is hallucination detection, defined as the confident generation of false or unsupported content. Surveys distinguish between *factual* and *faithfulness* hallucinations and position them as primary quality defects rather than minor inaccuracies [163] [164]. Empirical audits report persistent hallucination rates across models, including fabricated citations and incorrect medical advice[165]. Emerging approaches treat hallucination detection as an *uncertainty estimation*

[152] Tosi, G., Di Lascio, F.M.L., & Morisio, M. (2024) *Studying the quality of source code generated by large language models*. *Future Internet*, 16(6), 188. MDPI.
Available at: https://doi.org/10.3390/fi16060188
[153] Cotroneo, D., De Simone, L., Pietrantuono, R., & Russo, S. (2024) *Automating the correctness assessment of AI-generated code*. *Journal of Systems and Software*, 212, 111995. Elsevier.
[154] Sabra, M., Liu, Y., Liu, J., & Shen, Y. (2025) *Assessing the quality and security of AI-generated code: A quantitative analysis*. *arXiv preprint*, arXiv:2508.14727.
[155] Kirchner, A., & Knoll, A. (2025) *Generating automotive code: Large language models for software development and verification in safety-critical systems*. *arXiv preprint*, arXiv:2506.04038.
[156] Pan, X., Hentges, J., Zeller, A., & Knoll, A. (2025) *Automating automotive software development: A synergy of generative AI and formal methods*. *arXiv preprint*, arXiv:2505.02500.
[157] McKinsey & Company. (2025) *From engines to algorithms: GenAI in automotive software development*. McKinsey Center for Future Mobility.
[158] Liang, G., et al. (2024) ‘Evaluating Code Generation with Large Language Models’, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
[159] Sabra, A., Schmitt, O. & Tyler, J. (2025) ‘Assessing the Quality and Security of AI-Generated Code: A Quantitative Analysis’. arXiv preprint arXiv:2508.14727.
[160] Kirchner, A. & Knoll, A. (2025) ‘Generating Automotive Code: Large Language Models for Software Development and Verification in Safety-Critical Systems’. arXiv preprint arXiv:2506.04038.
[161] Liu, M., et al. (2024) ‘An Empirical Study of the Code Generation of Safety-Critical Software’, *Applied Sciences*, 14(3), 1046.
[162] Li, W., Gao, K., He, H. & Zhou, M. (2024) ‘LiCoEval: Evaluating LLMs on License Compliance in Code Generation’. arXiv preprint arXiv:2408.02487.
[163] Sahoo, P. et al. (2024) ‘A comprehensive survey of hallucination in large language models’, *Findings of EMNLP 2024*.
[164] Huang, L. et al. (2025) ‘A survey on hallucination in large language models’, *ACM Computing Surveys*.
[165] Rahman, A.B.M.A. et al. (2024) ‘DefAn: Definitive Answer Dataset for LLMs Hallucination Evaluation’, *arXiv:2406.09155*.

problem, using entropy-based confidence measures to flag unreliable responses[166]. Overall, hallucination control and calibration are now central to quality assurance in GenAI systems.

[166] Farquhar, S. et al. (2024) ‘Detecting hallucinations in large language models using uncertainty estimation’, *Nature*.