The Journal of Systems and Software 210 (2024) 111944

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A survey of energy concerns for software engineering ☆

Sung Une Lee [a,*], Niroshinie Fernando [b], Kevin Lee [b], Jean-Guy Schneider [c]

[a] CSIRO's Data61, Clayton, 3168, VIC, Australia
[b] School of Infomation Technology, Deakin University, Geelong, 3220, VIC, Australia
[c] Faculty of Information Technology, Monash University, Clayton, 3800, VIC, Australia

ARTICLE INFO

Keywords:
Software engineering
Energy
Green
Sustainability

ABSTRACT

There is growing attention to energy efficiency in the software engineering field. This has been driven by modern technologies, for example, Internet of Things (IoT), Social Networking Services (SNS) and quantum computing. In addition to this, recent trends and concerns such as Environment, Social, and Governance (ESG) and human/societal/environmental well-being for responsible Artificial Intelligence (AI) have accelerated the use of energy efficient software. Despite this, energy concerns in this field have been less explored and studied. This limitation results in falling short to address and overcome greenability issues at the software level, and leaving critical challenges to be solved in this space. This study aims to address this limitation and fill the gap between previous studies. We survey green in software engineering framed by the ten knowledge areas of software engineering to not only cover the entire development life-cycle but also widen the scope of discussion to software process, method, and model management. Based on our comprehensive investigation, we discuss open challenges, trade-offs and implications of this study for both researchers and practitioners.

1. Introduction

Ever increasing data and IoT devices require more computational power and storage space, resulting in growing concerns about green/energy-efficient IT in recent years. The energy consumption of IT, for example, data centers, has been dramatically increasing every year (Naumann et al., 2011). This has led to increased global carbon emissions, set to jump by 1.5 billion tonnes in 2021 according to the global energy review 2021 by International Energy Agency (IEA) (International Energy Agency, 2021). In addition, mobile computing platforms such as smartphones and tablets and IoT sensors are everywhere in our lives. Unfortunately, they are all battery-driven, and thus energy-constrained (Pinto and Castor, 2017). It draws more attention to software energy efficiency which has become a primary concern (Georgiou et al., 2019a).

The focus of researchers, however, has been mainly on integrated systems or hardware. This results in limited attention to software (Mahmoud and Ahmad, 2013; Pinto and Castor, 2017). Furthermore, previous studies on energy and software engineering have addressed specific issues such as energy metrics, and tools and techniques for measuring energy consumption of software. Researchers have proposed green software processes and models, yet they still remain at the conceptual level. Georgiou et al. (2019a) addressed open research questions for energy-efficient programming, and introduced existing approaches, tools, and techniques. The authors focused on the particular software model (Waterfall), and therefore limited the scope of their study. Energy concerns should be considered for end-to-end management of best practices, this limitation may fall short for the operationalization of green software engineering in practice.

Existing studies on energy efficient software engineering provide a good starting point. Yet, there is lack of information to provide a wide spectrum of insights across the entire discipline areas of software engineering and to guide researchers and practitioners in real-world contexts.

In this study, we aim to address this gap, by a broad-spectrum investigation of energy concerns in software engineering. As analysis dimensions, ten areas of Software Engineering Body of Knowledge (SWEBOK) are chosen: software requirements, software design, software construction, software testing, software maintenance, software configuration management, software deployment, software process, software models and methods and software quality. The selected areas are essential to cover the entire life-cycle of software development and enable this study to articulate new insights and trade-offs of end-to-end energy concerns in any software development methods including Waterfall, Agile and DevOps.

This study significantly contributes to the field by providing a comprehensive understanding of how energy concerns are addressed

☆ Editor: Patricia Lago.

* Corresponding author.
E-mail address: sunny.lee@data61.csiro.au (S.U. Lee).

https://doi.org/10.1016/j.jss.2023.111944
Received 24 February 2023; Received in revised form 1 November 2023; Accepted 22 December 2023
Available online 2 January 2024

in software engineering. We offer a broad overview of energy considerations, along with discussions on relevant tools and techniques for software engineers. The identification of open challenges and trade-offs across various key software engineering areas enhances our understanding of the current state of green software engineering. Additionally, we propose practical and concise approaches, including a one-page green software engineering framework, a goal model for addressing challenges in green software engineering, and green elements for incorporating energy efficiency into AI systems within the supply chain.

The remainder of this paper is as follows. The next section provides background information of green software engineering. Section 3 describes our approach for investigation. We then present the results of our analysis in Sections 4, 5 and 6. Section 7 identifies key challenges, limitations and gaps and trade-offs for the next steps. This section also provides the practical implications of this study. We conclude this study and discuss the future research directions in Section 8.

2. Green and software engineering

Energy concerns in software engineering have gained significant attention due to the growing recognition of the environmental and economic impact of software systems. The concept of sustainability and greenability has been frequently discussed in relation to energy concerns in software engineering (Calero et al., 2014). *Sustainability*, as defined by Brundtland Commission (1987), involves meeting the needs of the present without compromising the ability of future generations to meet their own needs. It encompasses three essential dimensions: social, economic, and environmental. Environmental sustainability focuses on protecting the environment and conserving resources, while social sustainability aims to ensure fairness, inclusivity, and equal access to resources and opportunities. Economic sustainability seeks to achieve sustainable economic growth that balances environmental and social well-being. By integrating these dimensions, a more equitable and prosperous future can be created.

Software sustainability is often defined as "the ability of a socio-technical system to persist over time" (Becker et al., 2015), and is sometimes described in the literature as a non-functional requirement or software quality (Penzenstadler et al., 2014a). However the term sustainability as relevant to software is more nuanced, and there is no unified understanding within the software engineering domain, with different researchers providing different perspectives (Venters et al., 2018, 2021, 2023). For example, Penzenstadler suggests that software sustainability to be defined as preserving the function of a system over a defined period of time, via considering three variables of system, function, and time (Penzenstadler, 2013). Some researchers present the concept of sustainable software as to minimize the negative impacts on the society, the economy, and the environment throughout the software development life-cycle (Dick et al., 2010; Calero et al., 2014). This involves reducing resource consumption, such as power usage, to mitigate environmental impact and promote social and economic sustainability (Calero and Piattini, 2015). Hilty, Lohmann, and Huang offer insights into the domains of ICT geared towards sustainability, highlighting Environmental Informatics, Green IT, and Sustainable Human-Computer Interaction (Hilty et al., 2011). They argue that mere technological efficiency will not yield sustainability due to challenges like Jevon's paradox (Alcott, 2005). Instead, sustainable growth demands a blend of efficiency and sufficiency, notably by separating economic expansion from environmental repercussions and natural resource consumption. In addition to the three sustainability dimensions of social, economic and environmental, some studies have addressed additional aspects, such as human sustainability and technical sustainability (Alharthi et al., 2019; Penzenstadler and Femmer, 2013). Human sustainability focuses on ensuring the well-being and development of individuals and communities, addressing social issues, and promoting a high quality of life (Goodland et al., 2002). For example, designing software that is inclusive and accessible ensures that diverse populations can benefit equitably from technological advancements. Technical sustainability, on the other hand, emphasizes the long-term usage and adequate evolution of software systems, with a focus on developing and implementing technologies with minimal environmental impact to support sustainable practices (Calero and Piattini, 2015). For example, building software that is modular and maintainable ensures its longevity and adaptability to future technological shifts. The effects in one dimension can influence others, such as environmental impacts affecting individual, societal, and economic aspects. For example, from an environmental perspective, cloud computing can reduce the need for physical hardware, leading to decreased energy consumption and e-waste. Economically, businesses can achieve cost savings through scalable cloud solutions, avoiding the overhead of maintaining and updating on-premises hardware. Societally, cloud applications can democratize access to advanced software tools, as they are often available on a wide range of devices and can be accessed from anywhere, promoting digital inclusion. However, the technological dimension presents challenges such as potential security vulnerabilities and the need for reliable internet connectivity. Despite this interdependence, these dimensions offer a valuable framework for examining pertinent issues (Becker et al., 2015).

To assess and evaluate the sustainability and energy concerns in software engineering, the concept of greenability has been introduced (Calero et al., 2013). *Greenability* is a software quality characteristic based on the ISO/IEC 25000 software product quality standard (Calero et al., 2013; Calero and Piattini, 2015). It encompasses software product characteristics including energy consumption, resource optimization, capacity optimization, and perdurability. Greenability also considers *quality in use* characteristics such as efficiency optimization, user's environmental perception, and minimization of environmental effects.

Green software can be categorized into two concepts depending on the type of contribution: green in software and green by software (Erdelyi, 2013). *Green in software* refers to software itself reducing energy consumption to become greener by running on environmentally friendly way. Green in software engineering is regarded as part of green in software, which seeks to integrate environmentally sustainable practices into the software development process and other related activities within the field of software engineering (Calero and Piattini, 2015).

Green by software, on the other hand, generally implies that software supports green philosophy, aiming at producing as little waste as possible by means of software (Erdelyi, 2013). This includes sufficient process control, replacing environmental harm activities by green ones, education and training, and information delivery.

This study primarily focuses on the former, *green in software*, to gain an in-depth understanding of how energy concerns are addressed in software engineering and facilitate the development of energy-efficient software.

Berkhout and Hertin propose a three-tiered framework for assessing the sustainability of Information and Communication Technology (ICT) systems, called *orders of effects* (Berkhout and Hertin, 2001). The first tier, known as *first-order effects*, deals with the direct environmental consequences stemming from the ICT hardware's lifecycle. These effects encompass impacts such as (i) manufacturing impact which refers to the environmental toll of producing the hardware, including resource extraction, energy consumption, and waste generation, (ii) operational energy use which refers to the energy consumed by the ICT hardware during its operational phase, and (iii) electronic waste, which refers to the waste generated at the end of the hardware's lifecycle, including disposal and recycling challenges. Software engineers can contribute to mitigating these effects by developing software that is optimized for energy efficiency, and designing software to extend the lifespan of hardware through efficient resource utilization, reducing the frequency of hardware replacements and, consequently, electronic waste. The second tier, known as *second-order effects* refer to the indirect

environmental impacts that ICT systems exert on other processes or systems. Examples include the optimization of traffic management systems, improvements in industrial processes, and the facilitation of remote work. Software engineers can play a crucial role in minimizing adverse second-order effects by considering these indirect impacts on other systems, by understanding the context of use and how users interact with the software system (Penzenstadler et al., 2014b). The third tier, known as *third-order effects* refer to the long-term, indirect consequences that arise from ICT usage. These can include economic growth, lifestyle changes, and rebound effects where initial environmental savings are offset by increased consumption. For example, a software system that makes online shopping more efficient could inadvertently lead to increased consumerism and, consequently, more waste and emissions. Interdisciplinary collaborations amongst software engineering, and other fields like economics, sociology and ethics are required to gain insights into these effects (Penzenstadler et al., 2014b). In this paper, we mainly consider the *first-order effects* of software systems in terms of energy consumption only.

3. Methodology

3.1. Software engineering body of knowledge (SWEBOK)

Software sustainability should be considered throughout the entire software development life-cycle and processes to facilitate continuous "green in software" practices. This concept aligns with the principles of SWEBOK, an international standard (ISO/IEC TR 19759:2005) that provides a comprehensive guide to the generally accepted software engineering body of knowledge (Bourque et al., 1999). SWEBOK is widely utilized for research and education purposes, offering a broad range of knowledge areas that serve as the foundation for holistic analysis in the field of software engineering. SWEBOK provides 15 distinct software engineering areas that collectively cover the knowledge and practices. These areas provide a comprehensive framework for understanding and addressing various aspects of software development and management.

As mentioned, we selected ten areas based on SWEBOK considering their direct relevance and impact on energy consumption and sustainability within the software development process. The selected areas directly address aspects of software engineering that have a significant influence on energy consumption and environmental impact. Additionally, they cover a wide range of activities and processes involved in software engineering, ensuring that various dimensions of energy concerns are addressed. By examining these specific areas, we can identify potential energy optimization opportunities, highlight existing challenges, and propose recommendations for energy-efficient software engineering practices. While the excluded areas such as software engineering management, professional practice, economics, and the foundational aspects (computing, mathematical, and engineering foundations) areas are also important in the broader context of software engineering, their direct impact on energy consumption and sustainability may not be as prominent as in the chosen areas.

The ten knowledge areas encompass the entire life-cycle of software development and incorporate management areas such as software quality and process and models and methods, ensuring a comprehensive examination of energy concerns in software engineering.

Software requirements. This knowledge area focuses on the identification, analysis, specification, and validation of software requirements, as well as providing guidelines for requirement management. It addresses the process of identifying energy concerns as software requirements and discusses how these requirements should be managed throughout the software life-cycle.

Software design. This area encompasses software architecture, components, and other system characteristics. It involves developing models and representations that align with the software requirements and constraints. In the context of energy concerns, it examines how software design addresses green requirements and discusses its impact on software energy consumption.

Software construction. This area involves the detailed creation of functional software through coding, verification, and testing activities. It includes applying coding guidelines, verification techniques, and testing methodologies. In the context of energy concerns, it explores how energy efficiency is supported during these stages and examines the energy impacts of different development environments, structures, and strategies.

Software testing. This area involves the process of systematically evaluating software to ensure its quality and functionality. It includes various testing techniques, such as unit testing, integration testing, and system testing. This study explores how energy consumption can be measured and analyzed during testing activities. It also addresses the inclusion of energy-related test cases and the evaluation of energy efficiency in software through proper analysis of energy usage changes and identifying potential solutions to optimize energy consumption during testing.

Software deployment. This area deals with activities related to software release, installation, and distribution. Although SWEBOK does not have a separate "Software deployment" area, we have included it in this study to investigate the activities and considerations related to the release and distribution of software, specifically focusing on energy efficiency during the deployment phase, which is an important aspect of green software engineering.

Software configuration management. This area provides techniques and tools for managing the functional and physical characteristics of software. It includes version control, change management, and configuration management. In the context of energy concerns, it addresses how different software versions and configurations influence energy efficiency and how to manage software configurations to optimize energy consumption.

Software process. This area focuses on defining, managing, and improving software development processes. It comprises process models, methodologies, and best practices. In this study, it has a focus on how previous studies have addressed energy concerns to support energy-efficient software development processes.

Software models and methods. This area encompasses various models, methods, and techniques used in software engineering. It includes software development methodologies, modeling languages, and software engineering tools. In the context of energy concerns, it examines how energy-aware practices and techniques can be integrated into software development models and methods.

Software quality. This area emphasizes the measurement and assurance of software quality, reflecting desirable product characteristics. It includes quality assurance processes, software testing, and quality metrics. In the context of energy concerns, it highlights the inclusion of energy concerns as software quality requirements and discusses energy metrics and supporting approaches for measuring software quality.

3.2. Literature review

In this study, we conducted a literature review to identify theoretically and practically important aspects of energy concerns in software engineering as discussed in academic literature. We followed the literature review technique from Webster and Watson (2002) to conduct the review. While our approach draws inspiration from the methodology proposed by Webster and Watson, it should be noted that it is not a strictly rigorous process like a traditional systematic literature review. However, we incorporated certain systematic elements in our approach to ensure a comprehensive and organized review of the relevant literature. Drawing inspiration from Kitchenham's approach (Kitchenham et al., 2009), we have formulated well-defined research questions, established a transparent and reproducible review process, and documented details such as the search strategy, inclusion and exclusion criteria, and the protocol for study selection.

To explore a broad scope of the topic, we formulated three research questions as follows:

Table 1
The overview of a literature review.

Category	Description
Technique	The literature review technique from Webster and Watson (Webster and Watson, 2002).
Data collection	Literature data collection, data selection, and data evaluation.
	* conducted: 2021.04 – 2022.02
Search keyword	10 keywords used for literature search.
	(energy OR green OR sustainability) AND software AND (quality OR metric)
	(energy OR green OR sustainability) AND software AND requirements)
	(energy OR green OR sustainability) AND software AND (design OR architecture OR model)
	(energy OR green OR sustainability) AND software AND (implementation OR development OR construction)
	(energy OR green OR sustainability) AND software AND (test OR testing)
	(energy OR green OR sustainability) AND software AND (deployment OR release OR installation)
	(energy OR green OR sustainability) AND software AND (configuration OR version OR change)
	(energy OR green OR sustainability) AND software AND (maintenance OR operation)
	(energy OR green OR sustainability) AND software AND (process OR practice)
	(energy OR green OR sustainability) AND software AND (model OR method)
Condition	Metadata: in all
	Time duration: 2010–2021
Database	ACM DL, IEEE Xplore and Springer Link
Search result	6,620,775 (Total)- ACM DL (685,108), IEEE Xplore (88,316), Springer Link (5,847,351)
Criteria	(Selection)
	Strong relevance to the ten dimensions
	Peer-reviewed articles and books
	Availability of full-text access
	Articles written in English
	(Exclusion)
	articles focused on hardware or high-level concepts
	Non-peer-reviewed/non-academic papers
	Pertained solely to specific business domains or technologies
	Duplicated articles
Protocol	1. Divide the ten knowledge areas among the four authors.
	2. Manual evaluation by each author, based on the inclusion and exclusion criteria.
	3. Review and cross-check by all authors.
	4. Agreement and final decision by all authors
Final result	Selected number: 131 → Final number: 101 (after removing duplicates)

- RQ1. What energy concerns have been addressed within the ten areas of Software Engineering Body of Knowledge (SWEBOK) dimensions?
  - RQ 1.1. What energy concerns have been addressed in the Software Development Life Cycle (SDLC), including the stages of Software requirements, Software design, Software construction, Software testing, Software deployment, and Software configuration management?
  - RQ 1.2. What energy concerns have been addressed in Software quality area?
  - RQ 1.3. What energy concerns have been addressed in Software Processes, models and methods?
- RQ2. What are the key challenges faced in incorporating green into the software development process?
- RQ3. What are the practical implications of integrating energy efficiency considerations into software engineering practices?

Table 1 shows the overview of the literature review conducted in this study.

The initial step of the literature review involved selecting appropriate keywords (Fig. 1). We utilized ten dimensions from SWEBOK and formulated search queries based on these keywords. During this step, we surveyed the state-of-the-art research published between 2010 and 2021. Our choice of this time frame is driven by the growing attention given to energy concerns in software, both within industry and academia since 2010 (Georgiou et al., 2019a).

The queries were utilized to search for relevant works in digital libraries: ACM DL, IEEE Xplore, and SpringerLink. Our initial search yielded a significant number of records in these databases: 685,108, 88,316, and 5,847,351 records, respectively. However, due to time and resource constraints, we were unable to include other databases such as Scopus and ScienceDirect.

In addition to the venue-based search strategy, we employed Google Scholar as a supplementary tool to identify additional relevant papers. This approach was not aimed at traditional snowballing through forward or backward searching but rather involved a targeted inclusion of papers found in citations within the selected venues.

In the next step, we conducted a screening process by assessing the titles and abstracts of the initial search results. Given the large number of results, we focused on these 2000 articles to identify the most relevant ones for further investigation We then performed a full-text assessment of these articles, applying the selection criteria defined in our research protocol. The selection criteria encompassed the following aspects: (i) explicit mention of software and energy concerns with strong relevance to the ten dimensions (relevance), (ii) articles published in peer-reviewed journals and conferences, and book chapters (quality), (iii) availability of full-text access, and (iv) articles written in English (language). Conversely, we excluded articles that (i) primarily focused on hardware or presented high-level concepts (e.g., overviews, strategies), (ii) were non-peer-reviewed and non-academic papers, (iii) pertained solely to specific business domains or technologies, and (iv) were duplicates.

The ten knowledge areas from SWEBOK were distributed among the four authors. Each author conducted a manual evaluation of papers relevant to their assigned knowledge area, adhering to the inclusion and exclusion criteria. For example, the first author evaluated *Requirements,*

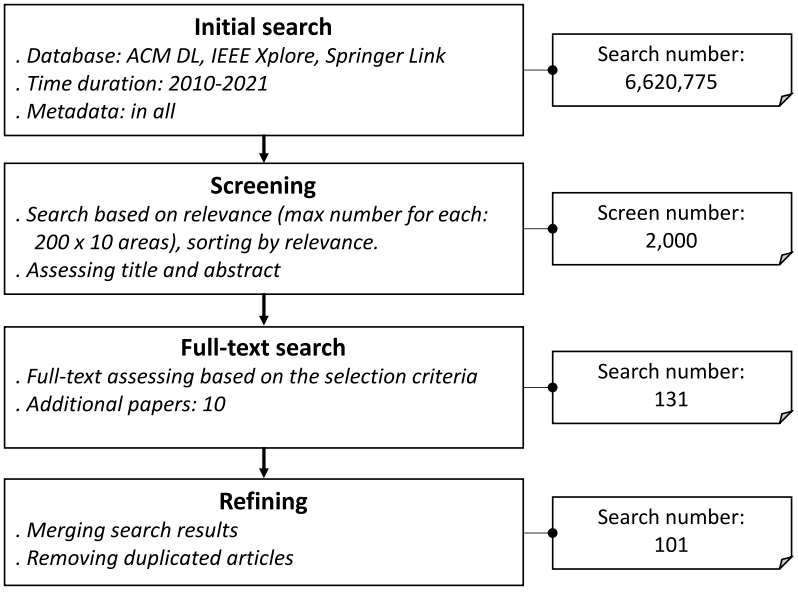


Fig. 1. The literature review flow to search articles and the results.

Design, Process and Models/Method, and the second author evaluated *Construction and Testing*. The evaluation results were then reviewed and cross-checked collaboratively by all authors to ensure consistency and accuracy in the selection process. In case all authors do not agree, the final decision was taken based on the majority voting mechanism. This initial evaluation involved filtering papers based on titles and abstracts to identify those most pertinent to the respective knowledge area. As a result, we identified 131 articles[1] that met the inclusion criteria (Table 2). Lastly, we removed any duplicated articles from the search results, resulting in a final selection of 101 unique articles for analysis.

After selecting the final articles, each article underwent a thorough review and coding process. We examined the selected articles and identified key information related to energy concerns for each dimension of software engineering and discussed in the following sections. The coding process involved categorizing the articles based on their research focus, findings, challenges and implications for energy efficiency. Any discrepancies in coding were resolved through discussion and consensus among the reviewers. This rigorous coding process ensured the reliability and validity of the data extracted from the selected articles. Table 2 presents the key considerations for each area and the papers analyzed in this study.

Fig. 2 provides an overview of how the selected articles are distributed across different software engineering domains and highlights the types of articles included in the study.

In Fig. 2(a), the bars are color-coded to differentiate between different article types, such as conference papers, journal articles, and book chapters. This color coding helps to visually distinguish the types of articles included in the study. Among the selected articles, the area of construction emerged as the most prominent, with a total of 31 articles selected for analysis. This indicates a substantial focus on energy concerns in construction-related topics in the software engineering field. On the other hand, we have found the least articles related to configuration and models and methods; with only 7 articles selected for analysis. This may highlight potential research gaps or areas that warrant further exploration.

Fig. 2(b) represents the distribution of article types without considering software engineering areas. It is important to note that the figure was generated after removing duplicated articles (30 articles), ensuring that each selected article is unique and contributes distinct findings to the study. The figure reveals that conference papers have the highest representation, comprising 58 articles, followed by journal articles with 34 articles, while book chapters have the lowest representation with 9 articles. This distribution underscores the prominence of conference papers as a prevalent source of research in the analyzed literature.

Furthermore, Fig. 3(a) illustrates the distribution of selected articles by year, primarily covering the period from 2010 to 2021, with additional articles included through snowballing. The data reveals a peak in the year 2013 (with a peak average in 2014). However, the number of articles gradually decreased in subsequent years, indicating a shift in research trends within the field. Fig. 3(b) highlights this trend shift, with a significant decline in studies focusing on energy concerns in software quality, construction, testing and construction since 2014, while the area of software process has gained increased attention.

4. Energy metrics for software engineering

Software quality is a measure of how well software is designed in the software engineering process. Quality is based on one of more metrics that allow a judgment to be made of how software meets these metrics. For example, in programming, quality can be measured by the readability of the code, ease of maintenance, the memory usage or code complexity. Standardized approaches exist to evaluate a software product for quality metrics, such as analyzing the quality of code (Baggen et al., 2012).

To ensure quality for energy in the software engineering life-cycle means first defining metrics that can be used to measure energy and then creating or adopting approaches to use the metrics to judge quality. In this section, we address RQ 1.2 by first evaluating literature in the areas of energy metrics for software engineering, then evaluating literature that uses these metrics to make energy quality judgments.

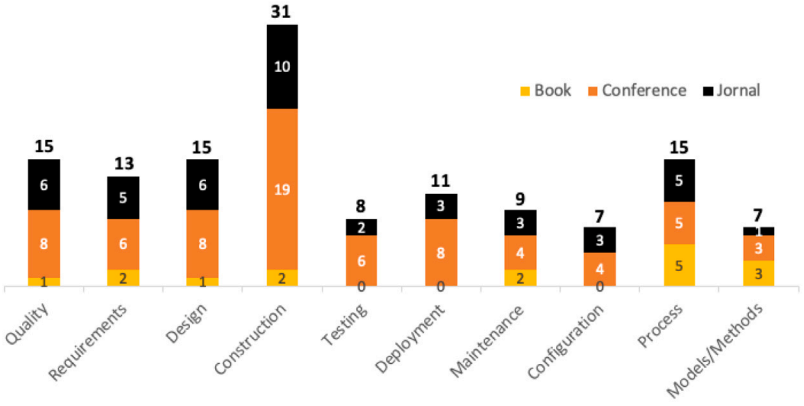
Energy metrics are most often discussed as part of sustainability metrics, which also include metrics such as carbon footprint, hardware obsolescence, organization sustainability and others (Naumann et al., 2011; Amsel et al., 2011). Most of these metrics do not apply to energy for software engineering. In sustainability, the metric of energy is normally simply for energy efficiency specified in energy per unit of work (Kern et al., 2013).

For software engineering, energy metrics are mainly focused on measuring the energy used for a particular operation. One such SDK

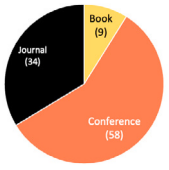
[1] https://docs.google.com/spreadsheets/d/1suDxN8OgR7tzyHqwuDUNHulg2pCBZFgS/edit?usp=sharing&ouid=101621542835091534934&rtpof=true&sd=true

Table 2
The key focus for surveying each area, and the relevant papers.

Category	Key consideration	Relevant paper
Requirements	Green requirements for software development and tools and techniques to aid requirements acquisition and management.	Calero and Piattini (2015), Bourque et al. (1999), Georgiou et al. (2019b), Taina (2011), Kern et al. (2013), Georgiou et al. (2019a), Mahmoud and Ahmad (2013), Saputri and Lee (2021), Roher and Richardson (2013), Condori-Fernandez et al. (2019), Manotas et al. (2016), Meridji and Issa (2013), Shenoy and Eeratta (2011)
Design	Previous studies on energy efficient activities at design phase and approaches to predict energy consumption.	Bourque et al. (1999), Mahmoud and Ahmad (2013), Saputri and Lee (2021), Agarwal et al. (2012), Chitchyan et al. (2016), Seo et al. (2008), Noureddine and Rajan (2015), Manotas et al. (2016), Lago et al. (2013), Sahin et al. (2012), Bunse and Stiemer (2013), Feitosa et al. (2017), Ampatzoglou et al. (2013), Fowler (1999), Chowdhury et al. (2019b)
Construction	Implementation level of software factors and environments increasing or decreasing energy usage.	Abdulsalam et al. (2014), Pereira et al. (2017), Couto et al. (2017), Oliveira et al. (2017), Corbalan et al. (2018), Abdulsalam et al. (2015), Yuki and Rajopadhye (2013), Pinto et al. (2014), Dukovic and Varga (2015), Raj et al. (2017), Sahin et al. (2016), Pinto et al. (2016), Hasan et al. (2016), Oliveira et al. (2021), Jagroep et al. (2017), Kazman et al. (2018), Manotas et al. (2016), Pang et al. (2015), Gottschalk et al. (2012), Park et al. (2014), Ardito et al. (2015), Morales et al. (2017), Vetro et al. (2013), Cruz and Abreu (2019), Siegmund et al. (2010), Pathak et al. (2012), Hao et al. (2013), Wilke et al. (2013), Hönig et al. (2014), Pereira et al. (2020), Manotas et al. (2014)
Testing	Useful techniques and strategies to conduct energy efficient testing.	Palit et al. (2011), Couto et al. (2014), Pereira et al. (2020), Banerjee et al. (2014), Chowdhury et al. (2019a), Jabbarvand et al. (2016), Linares-Vásquez et al. (2015), Hsu and Orso (2009)
Deployment	Important considerations and possible options for energy-saving deployment.	Johann et al. (2011), Naumann et al. (2011), Shenoy and Eeratta (2011), Hindle (2016), Al-Qamash et al. (2018), Chen et al. (2012, 2015), Hasan et al. (2019), Kwon and Tilevich (2013), Wu et al. (2017), Xing and Zhu (2009)
Maintenance	Key activities which should be implemented during maintenance for better green software engineering.	Bourque et al. (1999), Calero and Piattini (2015), Pérez-Castillo and Piattini (2014), Sahin et al. (2014), Mahmoud and Ahmad (2013), Penzenstadler (2012), Naumann et al. (2011), Johann et al. (2011), Shenoy and Eeratta (2011)
Configuration Management	Understanding of relationship between energy efficiency and software changes over time.	Hindle (2016), Dick et al. (2013), Zhang and Hindle (2014), Bangash et al. (2017), Hindle (2015), Sahar et al. (2019), Calero et al. (2021)
Process	Deep insights and understanding of green practices for software development process, and the application to different software methods.	Bourque et al. (1999), Naumann et al. (2011), Dick and Naumann (2010), Lami and Buglione (2012), Agarwal et al. (2012), Lami et al. (2012), Abdullah et al. (2015), Shenoy and Eeratta (2011), Anthony and Majid (2016), Dick et al. (2013), Naumann et al. (2015), Mahmoud and Ahmad (2013), Van Loon (2004), Kern et al. (2013), Saputri and Lee (2016)
Models/Methods	A set of activities for green software development when using different methods/models.	Lami et al. (2012), Chitchyan et al. (2016), Mahmoud and Ahmad (2013), Shenoy and Eeratta (2011), Dick et al. (2013), Naumann et al. (2015), Dick and Naumann (2010)
Quality	Efficient and effective ways of managing and measuring green quality of software.	Baggen et al. (2012), Steigerwald et al. (2008), Chatzigeorgiou and Stephanides (2002), Naumann et al. (2011), Amsel et al. (2011), Kipp et al. (2011), Kansal et al. (2010), Poess et al. (2010), Kern et al. (2013), Hogan (2009), Lange (2009), Kounev et al. (2020), Chen et al. (2012), Kiehne (2003), Broussely (2010), Ferreira et al. (2011)



(a) Number by software engineering area and article type: the grand total is 131.



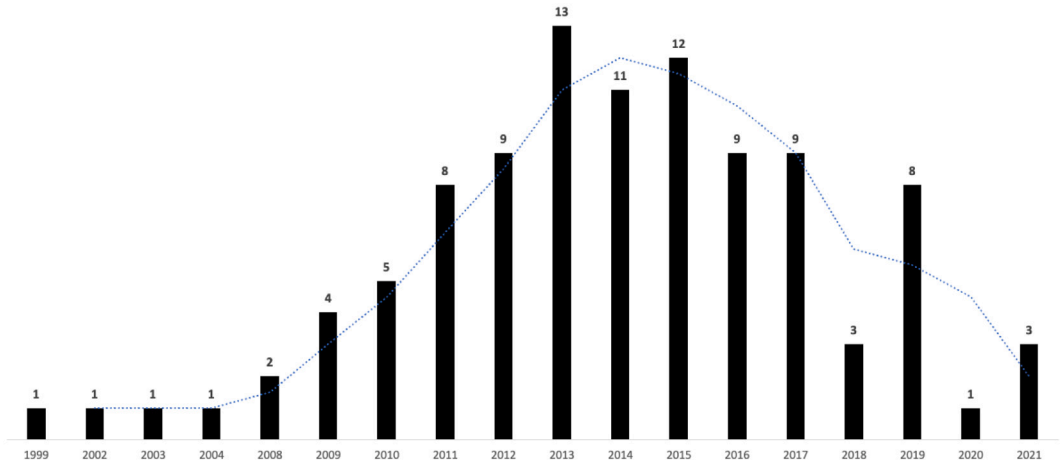
(b) Number by article type: after removing 30 duplicated articles, there are now 101 unique articles.

Fig. 2. The distribution of the selected articles in this study by the ten software areas and three article types.

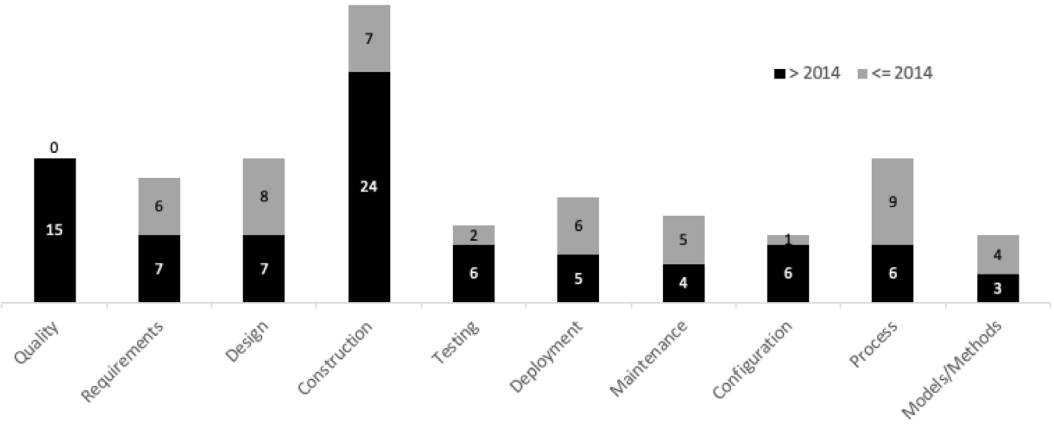
approach uses checkpoints in code to match program events to power readings, to produce correlated energy data (Steigerwald et al., 2008). Chatzigeorgiou and Stephanides (2002) defines processing power consumption to be the sum of the *base energy consumption* which is the energy consumption purely from the operating systems in question, and *overhead energy consumption* which is the additional energy use which bleeds from the system and from associated operations.

To provide useful energy metrics for the whole application lifecycle, Kipp et al. (2011) define the energy impact metrics to include (i) consumables generated, (ii) system power usage, (iii) supply chain caused by transportation and logistics. The increasingly common use of virtual machines for development, testing and deployment means that metrics can be used that include not just the running program, but the whole platform (Kansal et al., 2010). These approaches need to be followed actively to result in useful metrics.

As well as active approaches, there are attempts to benchmark systems as a whole to provide advice for businesses on their purchasing decisions (Poess et al., 2010). The Transaction Processing Performance Council (TPC) provides benchmarks of metrics for Online Transaction Processing (OLTP) systems (Hogan, 2009). The Standard Performance Evaluation Corporation (SPEC) provides benchmark approaches for power and performance of enterprise equipment (Lange, 2009). There



(a) Number by year; it shows the number of articles is decreasing since 2014.



(b) Number by year (comparison the number of articles before and after 2014).

Fig. 3. The distribution of the selected articles by year.

are also specific benchmark metrics for storage, including the SPC-1C and SPC-2C benchmarks (Kounev et al., 2020).

Measuring cloud computing poses additional challenges due to the user not having physical access to the resources. Chen et al. (2012) defines an energy consumption model that includes the energy consumption of storage, computation and communication for a particular task. The total of these for a task defines its total energy consumption.

For non permanently powered devices, the battery is of major concern. Generally the measures of importance are battery size, battery characteristics such as the chemistry choice and battery life (Kiehne, 2003). The priority of each of these depends heavily on the application. For electric and hybrid vehicles, the power-range (KW), energy-range (kWh) and voltage (V) range of the vehicle requirements in their respective units define the metrics for battery choice (Broussely, 2010). For smart phones and tablets, battery life is a ongoing concern, with a lot of effort being focused on improving this aspect, including analyzing user patterns (Ferreira et al., 2011).

Energy metrics allow data to be collected about energy in the software engineering life-cycle. To be useful, this data must be used to form a judgment on the quality of software in respect to energy. This judgment allows the product developer or software engineer to make decisions about things like the battery size.

Energy consumption over time is the most used energy metric for making judgments about the energy characteristics of hardware, software, device or activity. Generally, the assumption is that the more energy consumed over time, the higher the monetary cost, which is a undesirable characteristic.

The fine-grained energy consumption of software can be measured over time, such as for the interaction of CRM systems (Capra et al., 2012). The energy consumption cost of activities can also be quantified, such as the cost to reconfigure software (Ramachandran et al., 2015), the energy cost of executing scientific workflows (Warade et al., 2021), or the energy cost of calling web-services (Tanelli et al., 2008). For devices such as smart phones, energy efficiency can also be measured in energy consumption over time (Paul and Kundu, 2010).

5. Energy in the software engineering development life-cycle (SDLC)

Over the past decades, researchers and practitioners alike have devised a multitude of methods, techniques and tools to manage a variety of functional, non-functional and domain concerns within the context of the software development life-cycle (SDLC). These approaches have significantly improved the way the community thinks about software, how “expectations” are framed, how these expectations are translated into functioning software systems, and how the systems are ultimately modified in light of changing expectations. However, considerably less work has been done over the years to consider energy needs, constraints as well as energy footprints as primary considerations during the development life-cycle. In this section, we focus on RQ 1.1, and provide an overview of existing approaches that explicitly deal with energy concerns in SDLC as a primary driver in the corresponding methods, techniques or tools, respectively.

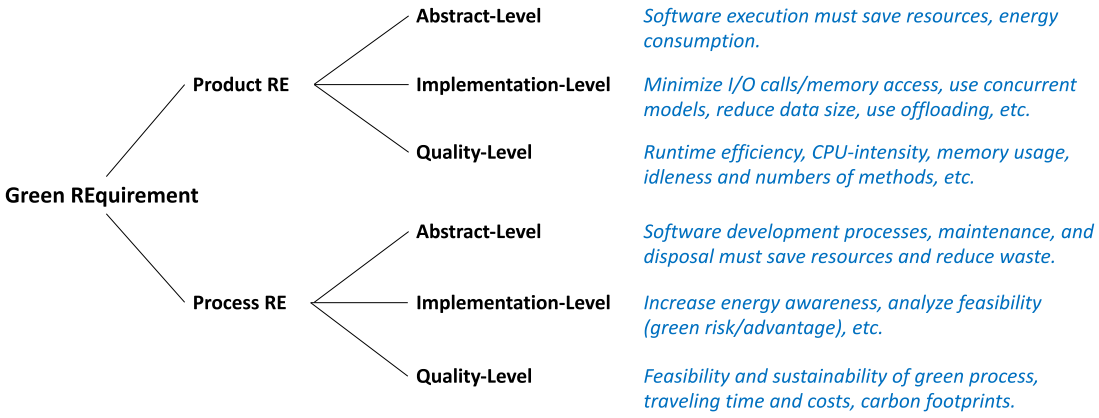


Fig. 4. Classification of Green Requirements.

5.1. Energy in requirements

Software requirements refer to the needs and constraints of software identified from various sources such as stakeholders, business rules, organizational and operational environment and regulations. Energy efficiency related software requirements (called as *green software requirements* in this section) are specifically related to the environmental impacts of systems (Calero and Piattini, 2015). This section introduces different types of green software requirements along with the roles and common examples. We also present supporting tools and techniques for the elicitation, analysis, specification and validation of the requirements.

5.1.1. Green software requirements

Software requirements are typically categorized into product and process requirements (Bourque et al., 1999). A product requirement represents a need or constraint on the software to be developed. Energy efficiency of product is regarded as a non-functional requirement (Georgiou et al., 2019b). A process requirement refers to a constraint on the development of the software (Bourque et al., 1999). The requirements are not directly related to software but influence the choice of technique, process, model and other development environments to improve energy efficiency.

Green software requirements are categorized into three types (abstract, implementation and quality) based on their role and characteristics of the requirements (Fig. 4). Abstract level requirements present the directions and ultimate goals of green software engineering (Taina, 2011). Implementation level requirements include actionable requirements which should be considered during software development. Quality level requirements enable stakeholders to understand the project is being managed and developed in energy-efficient ways.

A key abstract requirement for green software product is defined that, for example, software execution must save resources and reduce energy consumption and software must support sustainable development (Taina, 2011). Implementation requirements of product generally include technical considerations for green software: e.g., minimizing I/O calls, minimizing memory access, use of concurrent programming models, efficient data structures (less energy-greedy data structures), loop optimization (reduce control operations), reduction of data size (data compression), use of offloading methods (e.g., Cloud), consideration of approximate programming (reduce unnecessary precision of computations). Quality level requirements can be elicited based on the implementation level requirements. Accordingly, runtime efficiency, CPU-intensity, memory usage, idleness and numbers of methods can be considered as common quality aspects for software.

Process requirements at abstract level include that the software development processes, maintenance, and disposal must save resources and reduce waste (Taina, 2011). It is represented as the feasibility of the software project, and shows how resource efficient it is. The implementation level requirements include, for example, increasing energy consumption awareness, analyzing feasibility/risks of the project for making decision about green/energy advantage, constraining the process model (e.g., time, infrastructure and compliance) (Georgiou et al., 2019a; Mahmoud and Ahmad, 2013; Calero and Piattini, 2015). To ensure the feasibility of green process, traveling time and costs, carbon footprints, energy consumption and waste (resources consumed) are generally measured and managed during the life-cycle of the development (Calero and Piattini, 2015).

5.1.2. Tools and techniques

Compared to traditional software requirements, green software requirements are still not general and common in most software development projects (Saputri and Lee, 2021). Due to the lack of experience and knowledge of green requirements, extra efforts and specific tools and techniques for requirements elicitation and management may be required (Roher and Richardson, 2013).

The concept of recommender systems can be adopted to advice and provide common and standard requirements. Roher and Richardson (2013) proposed a context-aware approach to recommend green requirements for better time-to-market. The recommender system is specifically to support eliciting green software requirements. It provides a list of requirement archetypes to be considered. A user can select and tailor them for specific domains.

Another attempt to elicit, analyze and validate green requirements was conducted by Georgiou et al. (2019a). The authors presented green requirement suggestions identified by surveying practitioners and empirical evaluation. Survey is useful to extract concerns, ideas and requirements related to green software development from a group of software engineers. The participating group comprises various stakeholders having a consistent body of knowledge and understanding the interaction of software and hardware. The authors noted that an empirical evaluation can be used to identify the characteristics of software, provide the energy efficient requirements, and evaluate the requirements. A type of crowd-sourcing namely niche-sourcing is suggested by Condori-Fernandez et al. (2019). This approach is specifically useful to gather people that can contribute from multiple perspectives and characterize green requirements and their dependencies.

How to elaborate green requirements is addressed by Calero and Piattini (2015). A few steps are introduced to support green requirement engineering, which includes analyzing comprehensive context, finding stakeholders and eliciting sustainability objectives, goals and constraints, deriving sustainable system vision and usage model and refining requirements.

5.2. Energy in design

Software should be designed aiming to use less energy and resources (Agarwal et al., 2012). To support this concern, we discuss important design principles and design patterns for energy-efficient software as follows.

Software *decomposition and modularization* can be considered as the first design principle as it generally increases the re-usability of software components. *Separation of interface and implementation* reduces changes, and ultimately leads to less energy consumption. This is critical when extracting and designing software features, especially, for feature composition to minimize the coupling of features and increase the portability and adaptability of software (Saputri and Lee, 2021). *Sufficiency and completeness* enable stakeholders to be sure the software design and other artifacts meet all the green requirements specified in the previous phase. This is critical because green software design must be driven or approved by the customer (Chitchyan et al., 2016).

A design pattern is known as a common solution to a common problem. Some patterns such as anti-patterns increase energy consumption; also, others reduce energy usage (Manotas et al., 2016; Noureddine and Rajan, 2015). Several studies demonstrate *Flyweight, Mediator, and Proxy* patterns positively influence energy consumption (Sahin et al., 2012); *Flyweight* is recognized as the most energy-efficient design pattern, followed by *Visitor, Proxy and Mediator* patterns. Meanwhile, *Decorator, Prototype and Abstract* patterns increase energy consumption when running embedded systems or mobile applications. Specifically, *Decorator* is known as the most energy-hungry design pattern (Sahin et al., 2012; Bunse and Stiemer, 2013).

Predicting energy consumption of software at design-time would benefit to save time and cost for changing software for energy efficiency in the construction phase. However, this topic has been addressed by only a few studies. Sahin et al. (2012) created a set of software diagrams such as class diagrams, sequence diagrams and object diagrams to measure the number of objects instantiated by a program and messages between objects. The authors compared the number of objects and messages measured before and after applying design patterns, which shows how the selected design patterns influence potential energy usage. They executed applications before and after applying each of the selected design pattern and recorded the power consumption in each instance. The absolute energy difference per iteration was small (0.0002 J to 0.8672 J). However, the experiments were run on a Spartan-6 single board computer, which is inherently a low-power system. In contrast, most applications would be run on computers desktops, laptops or servers, which will consume more energy. Moreover, the experiment used a minimal number of classes and actions to apply the design patterns in the implementation, which would not be the case in many real world applications. Furthermore, it is likely that in a real-world setting, an implementation of a design pattern will be executed millions of times. Hence, even a small energy difference per iteration can cause significant aggregated energy impacts. The authors made the interesting observation that, energy usage was not consistent within design pattern categories (Creational, Structural, Behavioral). There was an increase in energy consumption when there were more objects, and when more messages were passed between the objects. For example, applying the *Abstract factory* pattern increased the number of objects from 11 to 13, the number of messages from 7 to 12, and the energy usage by 22%.

Feitosa et al. (2017) provide a more fine-grained assessment of the energy effects of 2 GoF design patterns (Template Method, and State/Strategy) via their pattern-related methods. They conducted a crossover study to compare the energy usage of pattern solutions with alternative solutions (non-pattern). The experiment utilized two open-source software systems JHotDraw[2] and Joda Time.[3] The authors based the alternative solutions from literature (Ampatzoglou et al., 2013; Fowler, 1999). The authors deliberately omitted from using alternative solutions optimized for energy efficiency because the aim was to investigate widely used alternatives, rather than energy optimized non-patterns that are unlikely to be widely known by software engineers. The study also investigated the effects of pattern-related parameters, using 2 metrics, namely, SLOC (number of source code lines of a method), MPC (number of calls, within the method, to other methods). The results of their experiments showed that (1) for the Template Method, pattern solutions consumed more energy than the alternative solutions, further confirmed by method level analysis, (2) for State/Strategy, pattern solutions consumed more energy than the alternative solution, although method level measurements showed high standard deviation and error and hence warrants further investigation, and (3) both SLOC and MPC have effects on energy consumption. However, it is worth noting that results also suggested that the GoF patterns Template Method, and State/Strategy, may be more energy-efficient when used to implement complex functionality with longer methods and multiple method calls to external classes.

As well as GoF design patterns, architectural patterns can have an energy impact. For example, in Chowdhury et al. (2019b) the authors demonstrated that by replacing the Model-View-Controller architectural pattern, with the Model-View-Presenter (MVP) pattern, energy consumption can be reduced up to 30%. Unsurprisingly, it was found that increased number of event producers and increased event production rate is correlated with higher energy use. This also correlates to the findings regarding message passing between objects in Sahin et al. (2012). When using MVP, the authors (Chowdhury et al., 2019b) recommend that dropping to be the most energy efficient strategy, although, if dropping is not feasible, developers can use bundling, which can still give a significant energy saving.

In general, studies have shown that design patterns can effect energy. However, it is not recommended to predict energy consumption based purely on the design/architectural pattern. There are a number of variables that also need to be considered, including the number of objects, runtime iterations and code complexity. Hence, it is advisable to make decisions regarding energy consumption in the design stage, prior to development (Chowdhury et al., 2019b).

5.3. Energy in construction

Software construction involves the realization of business requirements and design specifications into working software. This section examines the implementation strategies & tools that can have an impact on the runtime energy efficiency of the software.

5.3.1. Impact of programming language & compiler

The choice of programming language and/or the compiler used has been experimentally shown to have a substantial impact on the energy usage of a program. Multiple studies show that in general, C/C++ is the fastest and most energy efficient language (Abdulsalam et al., 2014; Pereira et al., 2017; Couto et al., 2017), although there are a few exceptions. For example, experiments in Abdulsalam et al. (2014) show that although C/C++ was generally more energy efficient than Java, Java was more energy efficient in instances of extensive dynamic memory allocation and deallocation. Experiments carried out by Oliveira et al. (2017) using a testbed of 3 Android phones and 5 native applications showed that Javascript was more energy efficient in 75% of the benchmarks used, compared to Java and C++. Four of the native applications were reengineered via a hybrid approach, where the most CPU intensive parts (in Java) were replaced by a hybrid of Javascript and C++. Results showed a significant energy saving in the hybrid versions. In one case, the savings were as high as 100x. A similar study by Corbalan et al. (2018) investigated energy consumption of mobile applications developed using different development platforms such as Andriod SDK, Native (NDK), Cordova, Titanium and Xamarin. Their

[2] http://www.jhotdraw.org/
[3] http://www.joda.org/joda-time/

experiments showed significant variances of energy usage for the same application, as much as 5 times more, across different development platforms.

Whether the execution time of a program is proportional to its energy use has been a much debated question (Pereira et al., 2017; Abdulsalam et al., 2015; Yuki and Rajopadhye, 2013; Pinto et al., 2014). However, there is no simple answer. The execution time of a program is only one factor of its energy use, as can be seen by the equation: $Energy = Time\ x\ Power$.

Pereira et al. (2017) investigate this question by studying the runtime energy consumption of 27 programming languages, and observing the performance for ten different programming problems as defined in the Computer Language Benchmark Game (CLBG). Their findings show that whilst the premise that faster programming languages consume less energy holds for some cases, it is not always the case. The top five energy efficient languages were also the fastest (C, Rust, C++, Ada, Java). However, the results were mixed for the remaining 22 languages. In fact, a similar study (Couto et al., 2017) shows that some languages such as OCaml, Fortran and Lua are more energy efficient than performance efficient. These findings are echoed in Pinto et al. (2014), where the authors empirically demonstrate that faster does not always equate with energy efficient, for concurrent programs. In general however, compiled languages were the most energy efficient as well as the fastest, except in a few edge cases such a string manipulations via regular expressions.

5.3.2. Impact of code obfuscation

Although commonly done to prevent software piracy, code obfuscation can incur increased energy usage in software (Dukovic and Varga, 2015). This concern is investigated in Dukovic and Varga (2015), where the authors evaluate three commercial obfuscators on two benchmark programs, for three obfuscation techniques, based on the resulting load profile. The study found significant energy impacts due to code obfuscation, and also demonstrated differences between energy use in different code obfuscators. Similar results were obtained in another comparable study (Raj et al., 2017), which revealed lexical obfuscation to be the least energy effects. A more thorough study by Sahin et al. (2016) investigated how different obfuscations can effect Android apps, by empirically studying 198 obfuscated versions of 11 Android applications. Similar to Dukovic and Varga (2015), their study also demonstrated that obfuscation typically increases energy use of applications with statistical significance.

5.3.3. Impact of data structures

The effect of data structures on energy efficiency has been empirically investigated in several studies (Pinto et al., 2014, 2016; Hasan et al., 2016; Oliveira et al., 2021). In Pinto et al. (2014) Pinto et al. presented an empirical study on the energy consumption of the Java threadsafe collections, where it was found that the choice of thread management constructs can have impacts on energy consumption. Data locality was also found to have a significant effect on energy use. This work was further extended in Pinto et al. (2016), where the authors empirically investigated the energy usage of 16 Java collection implementations, grouped under lists, sets and maps. They found significant energy differences between some of the newer and older implementations of the same collection (up to 17% in real world benchmarks). The study also revealed that different functions of the same collection can also have different energy footprints. For example, in ConcurrentSkipListMap, the remove() function consumes x4 energy than the insert() function. In another study (Hasan et al., 2016) the authors created energy profiles of commonly used API methods for variants of three Collections datatypes of List, Map, and Set. These profiles were based on their experiments measuring energy consumption of programs using Collections instances from the Java Collections Framework (JCF),[4] Apache Commons Collections (ACC),[5] and Trove.[6] Their findings show that for some operations, such as insertion, the energy differences between the Collections can be significant. For example, JCF's LinkedList implementation was the most energy efficient List when insertions were performed at the beginning. However, Trove's ArrayList was the most energy efficient List implementation when insertions were performed at the middle and end. In general, the study found the most energy efficient Collections to be, Trove's TIntArrayList for lists, JCF's HashMap for Maps, and JCF's HashSet for Sets. However, these results concerning a few of the JCF collections are contradictory to findings in Oliveira et al. (2021) where the authors state that ArrayList, HashMap, and Hashtable, are not energy efficient and should be avoided. In their study, Oliveira et al. (2021) built application-independent energy profiles of Java collections by executing several micro-benchmarks. Next, they extracted information about how the target software systems used the selected collections, for example, regarding usage context and frequency, via static analysis. Finally, they combined the constructed energy profiles and results of the usage analysis, and provided recommendations about energy efficiency, and alternatives. They found that overall, their method only recommended the use of JCF collections in 10.4% of the cases. In all other cases, alternative implementations of the collections (such as ACC) were recommended. However, it must be noted that, for many instances of JCF collections such as ArrayList, the reason that it was rarely recommended was because in most cases where it would be the best option, it was already being employed. Interestingly, recommendations varied heavily across devices, even when executing the same application.

5.3.4. Strategies to support energy-efficiency during implementation

Evidence from case studies show that efforts during the development phase can positively impact the energy efficiency of the software product. For example, Jagroep et al. carried out an exploratory case study (Jagroep et al., 2017) on the effects of measuring energy usage of the product, and providing feedback about the products energy usage to the stakeholders throughout development iterations. Microsoft Joulemeter (deprecated later) was used to measure energy consumption, and surveys were used to measure the effects of feedback on energy awareness of the stakeholders. To communicate the power usage of the product, the authors created an energy dashboard, which visualizes key energy findings, such as the energy delta between two releases of the software. Findings indicate that energy aware development needs to be supported by organizational policy. The energy dashboard had low acceptance amongst the stakeholders, but was found to be useful in terms of quantifiably highlighting energy savings as a result of development efforts. In another study, Kazman et al. (2018) reported the experience on the design and development of an automated weather station. Due to the nature of energy constraints in the hardware, sensing and data transmission, energy efficiency was vital for this particular application. Energy consumption was periodically calculated (using actual measurements and mathematical formulae). The authors conclude that energy efficiency should be treated as a quality attribute, and that its possible to substantially improve an application's energy use via small efforts in experimentation and prototyping, and small design changes.

However, despite the potential to incorporate energy efficiency during the development phase, researchers suggest that many software developers lack the required skills for energy aware development (Manotas et al., 2016; Pang et al., 2015). It is therefore crucial to find ways to support software developers decision-making during development, with minimal reliance on additional hardware instrumentation for measuring energy.

[4] http://docs.oracle.com/ javase/8/docs/technotes/guides/collections/

[5] http://commons.apache.org/proper/commons-collections/source-repository.html

[6] https://bitbucket.org/trove4j/trove

Refactoring (code level guidelines) is proposed in a number of papers (Gottschalk et al., 2012; Park et al., 2014; Ardito et al., 2015; Morales et al., 2017) as a strategy to support energy efficiency. In the refactoring phase, the 'energy aware developer' identifies code patterns that lead to higher energy usage, i.e., 'energy code smells' (Gottschalk et al., 2012; Vetro et al., 2013). Code level guidelines for identifying and re-engineering energy code smells include, cleaning up dead code (eg. unread variables), managing thread cycles (eg. when a service restarts after explicitly killed by the user, i.e., immortals), loop monitoring, method in-lining, removing redundant storage of data, timely release of resources (eg. GPS) and reducing data transmission. Higher level refactoring recommendations include understanding the hardware energy requirements, focusing on common usage scenarios, starting the refactoring process from higher level constructs rather than at lower levels, However, it should be noted that some of the above refactoring strategies have the potential to decrease code maintainability and readability, such as when method in-lining is used (Gottschalk et al., 2012).

The opposite of 'energy code smells' can be 'energy patterns', which are strategies commonly used by software engineers to increase the energy efficiency in applications. Cruz et al. presents a catalog of 22 energy patterns (Cruz and Abreu, 2019) used in 1027 Android and 756 iOS apps, by mining sourcecode from GitHub repositories.

Ardito et al. (2015) further proposes 'self-adaptation' which focuses on enabling the same application to have multiple configurations. In this way, the application is able to 'adapt' to allow for the best tradeoff between application features and energy consumption.

Another strategy proposed is providing energy related frameworks, and libraries to aid software developers (Siegmund et al., 2010; Hasan et al., 2016; Couto et al., 2017). For example, Siegmund et al. proposes a set of energy optimization libraries, so that developers can use energy saving code components without needing to know about the energy optimization algorithms in-depth (Siegmund et al., 2010). Similarly, in Hasan et al. (2016) the authors provide per-method energy profiles to guide developers on the usage of Java Collections. Developers can use the profiles to estimate the energy footprint of each Collection instance and take decisions accordingly. Couto et al. (2017) proposes a green ranking for programming languages, based on their energy efficiency. The goal of the ranking is to provide a method to aid developers building energy efficient software.

A number of tools to aid software development, have also been proposed. The main goal of these tools is to model the energy consumption of software via source code analysis. Eprof Pathak et al. (2012) is one of the earliest tools that measure the energy consumption via system-calls. In eLens (Hao et al., 2013), the tool provides line-by-line code analysis. For a given source file, the tool ranks each source line according to its energy cost via information such as bytecode, and api calls from various hardware components. JouleUnit (Wilke et al., 2013) is a generic framework to monitor the energy usage of source-code focusing specifically on testing during development. It correlates workload profiles with power measurements to derive energy profiles at the method level. Although the framework was implemented in both Android and NAO robots platforms, the authors did not provide data on energy savings as a result of using JouleUnit during development. Similar to Eprof (Pathak et al., 2012), the Green Advisor tool (Aggarwal et al., 2015) also estimates energy hotspots via system calls in source code. Green Advisor is based on the authors previous work Green Miner (Aggarwal et al., 2014). Manotas et al. followed an autotuning approach with the SEEDS tool (Manotas et al., 2014), which provides energy profiling for Java Collections. Software developers must first set the application-specific search space, such as setting optimization parameters. SEEDS then searches through the source code to find where energy saving alternative Collections can be used. Next, SEEDS transforms the original code into multiple alternatives, and profiles each of the alternatives to find the most energy saving transformation. Hönig et al. (2014) propose 'proactive energy-aware programming', in which the authors developed a tool named PEEK (Proactive Energy-awarE development Kit), which first automatically provides an energy analysis at function-level of the source code. Next, the tool also provides energy optimization hints to guide developers. In a somewhat similar feature to SEEDS, PEEK allows the comparison of multiple versions of the same code for energy efficiency, via 'snapshot bundling'. However, unlike SEEDS, in PEEK, the software developer needs to choose and submit the multiple versions for profiling. SPELL (Pereira et al., 2020) is another tool proposed by Pereira et al. which assists software developers by detecting energy inefficient source code fragments, or 'energy leaks', both on a project level, and on a method level. SPELL ranks the source code fragments based on energy efficiency, via test case execution followed by statistical analysis. Evaluation results showed that when a group of 15 developers used SPELL to detect energy leaks, they were able to achieve an average of 43% increase in energy efficiency. Similar to SEEDS (Manotas et al., 2014), the CT+ tool (Oliveira et al., 2021) also provides energy profiling for Java Collections. However, CT+ employs static code analysis whilst SEEDS employs dynamic analysis, and does not consider the impact of multi-threading, unlike CT+. Although they use different techniques, the reported energy efficiency after using the tool is somewhat similar. Table 3 provides a summary of the proposed tools intended as aids for software engineers to develop energy-efficient software.

5.4. Energy in testing

The testing phase of the SDLC focuses on evaluating the implemented software components to investigate if the quality standards as defined in the specification are met. This section discusses energy-related testing methods found in related work.

5.4.1. Test cases for locating energy leaks

Palit et al. (2011) is one of the first to discuss energy test cases. They provided a methodology to define user level test cases to evaluate the energy cost of smartphone apps. The proposed method employs 'user settable parameters' such as brightness and modes of connectivity. A 'configuration' consists of a particular combination of these user settable parameters. A test case can then be a represented as a pair <input; expected output>, where the input consists of a 'configuration' and an application specific setting, and the output is the energy cost. Experimental results showed that the method can identify effects of the user settable parameters on energy, at application level.

Couto et al. (2014) proposed a technique which categorizes the energy use of Android source code at method level, via the execution of test cases. Each test case is executed twice, where in the first execution, the stack trace of each test is logged, and the energy values are then logged for each test in the second execution. The energy classification is done by correlating the stack trace with the energy values, and using thresholds. Depending on the results, methods are then classified as either red (most energy use), yellow, or green (least energy use). The authors conducted experiments to test the energy performance of an open-source Android app using the proposed method, and their results showed that the execution time is highly correlated to the total energy consumption of an application. However, the technique was not verified with real energy measurements.

Data about its effectiveness in optimizing the energy efficiency of a given software system was also not provided.

The SPELL toolkit (Pereira et al., 2020) proposed by Pereira et al. provides energy consumption measurements via the execution of test cases and a statistical method that is based on Spectrum-Based Fault Localization (SBFL). SBFL is a testing technique used to assist on the location of program bugs. SBFL represents the test case executions in matrix form (A). Each row in (A) represents a test case, showing the various code components (e.g., statements, classes, methods) involved in each test case, accompanied by a vector (e) which shows if each test case failed or succeeded. Analysis of (A) and (e) produces a ranking

Table 3
Tools to aid software engineers develop energy-efficient software.

Name	Measurement/ estimation technique	External methods used	Measured components	Target platform	Reported energy efficiency
Eprof (2012) (Pathak et al., 2012)	Via system-call-driven Finite State Machines	Android's Traceview (deprecated)	CPU	Android & Windows Mobile	20%–65%.
eLens (2013) (Hao et al., 2013)	Estimates energy use via energy models	LEAP power measurement framework (Singh et al., 2010), energy profiler SEEP (Hönig et al., 2012)	CPU, RAM, WiFi, GPS	Android	–
JouleUnit (2013) (Wilke et al., 2013)	Test case execution followed by energy profiling	Hardware-based profiler uses a Yokogawa WT210 power meter, & the software-based profiler uses device dependent power rate probes	CPU, WiFi, Screen brightness	Language agnostic, implemented for Android and NAO robots	–
PEEK (2014) (Hönig et al., 2014)	Uses a modular architecture with sourcecode separated from the hardware specific backends performing energy analysis.	Current mirror analog energy measurements, energy profiler SEEP (Hönig et al., 2012)	CPU	Language agnostic	25%
SEEDS (2014) (Manotas et al., 2014)	Via a search-based approach to find and create alternative versions to produce the most energy saving collections	LEAP power measurement framework (Singh et al., 2010)	CPU, RAM	Java's Collections API	2% to 17%
GreenAdvisor (2015) (Aggarwal et al., 2015)	Via the comparison of system calls between different versions of source code	The Rule of Thumb model and the GreenMiner testbed (Aggarwal et al., 2014)	Unclear	Android	–
SPELL (2020) (Pereira et al., 2020)	Test case execution followed by statistical analysis (spectrum-based fault localization)	Intel's RAPL framework	CPU, DRAM	Desktops: Tool is implemented in Java	15%–74%, average 43%
CT+ (2021) (Oliveira et al., 2021)	Via inter-procedural static analysis, and recommending the most efficient Collections implementation	Recommendations of Georges et al. (2007) for Java performance evaluation	Unclear	Java's Collections API	4% to 16%

of the code components according to their probability of having faults. SPELL extends SBFL by including triples in (A). Instead of just the code component in SBFL, each element of the (A) matrix in SPELL holds $E_{i,j}$, $N_{i,j}$, $T_{i,j}$ where E denotes energy consumption, N denotes the number of executions and T denotes the execution time of element (i, j) in the matrix. Unlike SBFL the (e) vector in SPELL is used to denote possible excess of energy consumption. To use SPELL, test suites constructed for usage scenarios are executed, which allows for the collection for component based energy consumption data in the (A) matrix. These form the input values for the excess energy calculation in (e), which finally, provides a ranking of code components with highest probability of energy leaks. The SPELL tool was empirically evaluated in a case study where 15 developers were asked to optimize the energy efficiency of a software system using the tool. The authors show that developers using SPELL were able to improve the energy efficiency by 43%.

5.4.2. Automatic test generation for energy concerns

Although it is possible to manually build repeatable test cases that correspond to energy measurements, this requires a lot of time and expertise. In contrast, automatically generated test cases can be comparatively efficient.

In Banerjee et al. (2014) present an automated test generation framework that detects energy hotspots/bugs in Android applications. For each test input, the framework captures a sequence of user interactions, such as screen taps, that can cause increased energy usage. The authors argue that defining an appropriate metric for system-resource utilization is critical to detect energy inefficiencies. They define system-resource utilization (U) as the weighted sum of the utilization rates of all major power consuming hardware components in a device (eg. screen, WiFi, Radio, GPS and CPU) in a given period. Energy-consumption to Utilization (E/U) ratio is given as the measure of energy-inefficiency. Therefore, a high E/U ratio indicates an energy-inefficiency. To detect energy inefficiencies in an application, the framework first generates and stores event traces in a database, based on event flow graphs which capture all possible user event sequences. Next, it systematically executes the collected event traces, and compares the E/U values pre and post execution, via statistical methods. An energy hotspot/bug is detected if there is a significant difference between the E/U values pre and post execution. Evaluation using 30 Android apps uncovered energy bugs in 10 apps and energy hotspots in 3 apps. However this technique only considers GUI-based events, and therefore may not be able to uncover all energy inefficiencies within an app.

Greenscaler (Chowdhury et al., 2019a) is an energy model for Android apps, which uses automatically generated tests and test selection heuristics to build and continuously update energy models. Greenscaler uses heuristics to select test cases that exploit different energy consuming hardware components. Although code coverage has commonly been used as a heuristic for test case selection, the authors have showed that it is not an efficient measure for detecting energy consumption. Instead, similar to Banerjee et al. (2014), Greenscaler also employs metrics based on resource utilization, namely, two resource utilization heuristics *CPU-utilization* and *estimated energy utilization*. For a given app, Greenscaler generates test cases consisting of different randomly selected adb events. It then runs all the generated test cases, and selects the one that maximizes a given heuristic. Next, the selected

test case is run and GreenScaler collects energy consumption measurements, system calls measurements, and other process counters. These measurements are then added to the training corpus, which are finally used to train the energy model. Evaluations results of these energy models showed an upper error bound of 10% when compared with the ground truths. When compared with manually written tests of 984 versions from 24 real world Android apps, the upper error bound of GreenScaler was less than 10%.

5.4.3. Test suite minimization for detecting energy bugs

Jabbarvand et al. (2016) proposed a test suite minimization approach for Android apps, based on the list of energy greedy APIs sugested by Linares-Vásquez et al. (2015). Test suite minimization is commonly employed by testers to eliminate redundant test cases based on a given criteria such as statement coverage (Hsu and Orso, 2009). However, the authors argue that, although a commonly used metric, structural coverage is not an adequate measure for test suite minimization when it comes to energy bugs. Instead, the authors propose a new coverage metric named 'eCoverage' that indicates how much energy-hungry code are covered by a test. This technique can reduce the search space of energy test cases, thus enabling energy bugs to be fixed with less effort and time. Experimental results using real-world apps showed that the proposed approach can reduce the size of test suites (on average 84% in using integer programming and 81% using a greedy algorithm), with only minimal negative effects (3%–4%) on the effectiveness of the test suite.

5.5. Energy in deployment

Software deployment refers to activities and processes regarding to software release and installation of updates, patches and new applications. The size of deployment packages influences the costs of network bandwidth and disk space for storage (Shenoy and Eeratta, 2011). The most important things, however, are the operational costs and energy usage of software for deploying and using. The energy consumption of software generally depends on the deployment context which includes data centers, mobile, and wireless sensors (Hindle, 2016). Most companies owned their IT infrastructure in the past, but now external data centers (e.g., cloud) have become good deployment options for minimum financial and management burden (Al-Qamash et al., 2018)

Cloud data centers, however, face challenges in energy from many power-heavy units and software services dynamically provisioned on virtual machines (Hindle, 2016). The energy consumption of cloud systems may depend on both the hardware environments and runtime tasks. Some researchers studied on what the energy consumption patterns of each task in the cloud are, and how different workloads/system configurations influence energy consumption. In Chen et al. (2012) the authors proposed a *Green Cloud Computing (GCC)* model and analysis tool. The model comprises two parts: (i) fixed energy consumption during idle time, and (ii) variable energy consumption on storage, computation and communication resources for cloud tasks. The authors measured and analyzed the energy consumption of data-intensive tasks, computation-intensive tasks and communication-intensive tasks. This attempt provides in-depth insights and understanding of the relationship between energy consumption and different types of tasks executing in the cloud systems. This study was further developed by Chen et al. (2015). They proposed an automatic performance and energy consumption analysis tool for cloud applications. The tool supports finding the best deployment configuration that maximizes energy efficiency while guaranteeing system performance of cloud applications. The practical implications of this study can be seen in the example usage. The performance and energy consumption data of the systems are periodically collected by the tool, and the data can be used for various analysis and visualized for practitioners.

Fog computing allows bringing the processing units closer to the end-devices to improve computation power and reduce task execution and processing time (Al-Qamash et al., 2018). Multiple fog nodes play a critical role in recognizing their processing needs to decide whether it should be processed locally, or sent to the cloud. This mechanism improves energy efficiency of the cloud, yet there can be drawbacks such as limited performance (e.g., latency and bandwidth and dependency on the cloud).

Edge has been regarded as a technology to improve low latency connectivity in the terminal layer. It is often described as a more localized version of fog and cloud computing as it is closer to the end user (Al-Qamash et al., 2018). The edge devices can be both data consumers and data producers by collecting data from the database in the cloud and sending them to the user, and also caching, processing and storing data at the edge. The mechanism reduces the overhead of communication and processing and network latency in the cloud. Its utility, however, is limited in terms of resources such as data storage, computing power, and energy. This issue becomes more critical when the edge nodes are placed on mobile devices. Mobile devices such as smartphones play multiple roles: sensing data from human activities and behaviors, and acting as a gateway to gather data from other sensors and devices and transferring the data to the servers (Hasan et al., 2019). The availability of mobile applications is affected by their battery power (Hindle, 2016). In prior studies, cloud computing has been adopted as a mean to reduce the energy usage of mobile applications. Kwon and Tilevich (2013) proposed *cloud offloading* to optimize the energy consumption of mobile applications. The solution is designed to automatically deploy parts of the functionality of a mobile application to the cloud based on the approximate amount of energy consumed by the CPU to execute the functionality of the application components. The energy consumption graph used in this study presents which components are the energy-intensive functionality, and supports making a decision to execute the components in the cloud. As for network communication, the graph also includes a potential energy usage to transmit the necessary data of the components to execute remotely. This is regarded as the key trade-off to determine whether the functionality of the components will be executed locally or remotely.

5.6. Energy in maintenance

Software keeps changing or evolving. Software engineers need to ensure that the software continues to satisfy user requirements preserving functionality and the greenability characteristic (Bourque et al., 1999). Another key consideration in this phase is the concept of *ecological debt* which refers to the cost of delivering green software (Calero and Piattini, 2015). Ecological debt must be recognized and quantified at the maintenance stage, and it should include the cost of refactoring the software in the future for better maintability.

There are some possible techniques to improve software greenability without changing functionality and decreasing maintainability. *Refactoring* is concerned with improving software quality including greenability and preserving the functionality (Calero and Piattini, 2015). It is the best choice for dealing with source code to promise better software quality, specifically, for existing software. However, a proper level of granularity of refactoring should be considered to avoid excessive communication traffic caused by too fine-grained modules. *Identifying of bad smells and anti-patterns* is also critical as they can cause quality problems. Refactoring code smells, however, can have both positive and negative impacts on energy efficiency. For example, refactoring *Lazy class and Data class* may positively influence energy consumption improving the greenability of the software. In contrast, refactoring large class and generating a number of small classes can lead to more energy consumption due to increased message traffic. Likewise, finding and solving anti-patterns (e.g., god class, spaghetti code) improve software quality, but the impact of refactoring some

anti-patterns may also influence software greenability (Pérez-Castillo and Piattini, 2014).

Sahin et al. (2014) examined the impact of refactoring of Java applications. The authors chose six commonly used refactoring techniques (i.e., convert local variable to field, extract local variable, extract method, introduce indirection, inline method, and introduce parameter object). The experiments were conducted executing nine Java applications having different sizes and characteristics. The results showed that the energy consumption of the applications always decreased after applying the extract local variable technique. In Calero and Piattini (2015) the authors emphasized the importance of the prediction of the impact of refactoring the anti-patterns. The authors gave some examples of the impact of refactoring; for example, *the blob* anti-pattern (one class contains most responsibilities while the others hold only data and small processing) may reduce software energy efficiency.

Software maintenance theoretically includes all activities of software development. It also includes continuous monitoring of software quality such as greenability and knowledge management until replacement by a new system (Penzenstadler, 2012). At this phase, there should be installation of software patches or updates, training of users in regard to proper software usage, training of employees to carry out their tasks faster. These entail lower power consumption and proper configuration of the software to consume less power (Naumann et al., 2011; Johann et al., 2011). More specifically, maintenance is given to junior staff who might not have experience and knowledge of the software and environments such as programming languages (Mahmoud and Ahmad, 2013). Thus it is critical to give maintenance staff tutorials and courses, which will speed the process of maintenance reducing cost and increasing energy efficiency. System documentation is also important to better manage software configuration and provide understanding of the code for faster maintenance. Using electronic documentation and not using reverse engineering are good practices as it leads to time- and power-consuming activities (Shenoy and Eeratta, 2011).

5.7. Energy in configuration management

Software configuration refers to the functional and physical characteristics of software (Bourque et al., 1999). It is important to test different versions of software if there is a change in energy efficiency (Hindle, 2016; Dick et al., 2013; Zhang and Hindle, 2014). In Hindle (2016) the author emphasized that different versions of software may provide the same functions but perform differently in energy consumption. Other features, for example, use of brighter colors can also influence energy efficiency. Configuration control board (CCB) should decide whether to accept or reject proposed changes to the software based on green performance. To this end, it is essential to identify the cause of the decreased energy efficiency and solve it. For example, in Dick et al. (2013) the authors identified the fact that a new sorting algorithm increases energy consumption. The issue was remedied by changing the algorithm back to quicksort.

This approach, however, is an expensive and naive solution as it requires lots of time and resources (Bangash et al., 2017). Alternatively, software metrics can be used to estimate the impact of software change on energy consumption. In Hindle (2015) the author proposed a methodology for measuring the energy consumption of a set of software versions, and discussed the relationship between energy consumption and software metrics. It comprises several steps such as choosing a software product and a context, deciding on measurement and instrumentation, choosing a set of versions, developing a test case, configuring the testbed, running tests for each version, compiling and analyzing the results. The multiple case studies revealed that the energy consumption of a software is not constant over time for various reasons such as performance optimization and new features.

Hindle (2015) used several types of software metrics such as Chidamber and Kemerer (CK) Java Metrics which can process the bytecode of Java class files, Churn measures such as added lines, removed lines and File Churn, and lines of code (LOC) to relate them to software energy consumption. Yet, there is not clear evidence of correlation between the metrics and the actual energy usage of the selected software due to some limitations, for example, insufficient variation in software metrics.

Bangash et al. (2017) and Sahar et al. (2019) included more metrics such as Fernando Brito-e-Abreu's MOOD metric suite and Martin's Package metric suite, in addition to the CK metrics used in Hindle (2015). The authors also considered the fact that energy consumption depends on test execution path and the entire program cannot be correlated with energy consumption of a specific test execution trace. The results show that all of CK and Martin metrics (except one, Lack of Cohesion On Methods) correlate with energy. Especially, CK's CBO(Coupling between Objects) has a large negative correlation with energy and DIT (Depth of Inheritance Tree) shows a large positive correlation with energy. The study, however, needs improvements by evaluating more applications of varying complexity.

In Calero et al. (2021) the relationship between software maintainability and energy consumption is examined. Several versions of Redmine (a project management web application) were used to test the actual energy usage and assess different measures such as the number of lines of code or the complexity of the software. The maintainability metrics include Total Lines of Code (TLOC), Cyclomatic Complexity (CC), the Percentage of Comments in the Code (PCC) and the Percentage of Duplicate Code lines (PDC). The authors selected four different versions of Redmine which include the same functionalities and each version is sufficiently separated. As a result, there is a significant correlation between TLOC and the energy consumption of the system (more specifically, the energy use of the processor and the total energy use of the computer) in which the software is being run. However, the other three metrics (CC, PCC and PDS), are not related to the energy consumption of the system.

6. Processes, models and methods for energy in software engineering

Software engineering process refers to a set of work activities for software development and maintenance that incorporate criteria to transform inputs into outputs (Bourque et al., 1999). There are increasing concerns about resources and energy consumption during this transformation (Naumann et al., 2011). In this section, we address RQ 1.3, and provide a comprehensive discussion of approaches for better greenability including relevant models, techniques and tools for management and assessment of software engineering processes in the way of energy efficiency.

6.1. Energy in process

In this section, we address a number of considerations such as which software engineering processes and practices should be considered and managed and what process measurement metrics, tools and techniques can be considered in managing a green software project.

6.1.1. The green software processes and practices

Green software process is performed to introduce and integrate the greenness culture in an organization developing software (Lami et al., 2012). A conventional project considers more financial, operational and technical aspects, but a green software project must assess the impact on the environment (Agarwal et al., 2012).

Lami et al. (2012) and Lami and Buglione (2012) proposed three core processes for green software: management process, engineering process and qualification process. The proposed processes are described with the purpose and outcomes (practices). The study, however, does not address who and when involves in the life-cycle of the software.

Mahmoud and Ahmad (2013) proposed a new process model including green requirements, design and implementation, testing, green analysis process, maintenance and disposal process. The authors also proposed a new hybrid model between sequential, iterative, and agile process, which enables practitioners to apply the model to various types of projects. Similarly, Shenoy and Eeratta (2011) provided a set of suggestions to refine software development phases such as requirement-gathering, design, implementation, testing, deployment, maintenance and retirement. It also includes "infrastructural concerns" such as rooms, stationary and "quality software". In this study, the needs for metrics to measure greenability is mentioned.

The Dick and Naumann's model (Dick and Naumann, 2010) includes software development phases such as requirements analysis, design, implementation, operation and maintenance. Innovative enhancements and elements such as Sustainability Reviews and Previews, Process Assessment, Sustainability Journal, and Sustainability Retrospective are introduced. The elements can be integrated with existing process models and/or added as a separated task into them. However, there is little considerations of who and how to implement the model. OpenUP and Scrum are used to demonstrate how to apply the model in practice, yet more detailed and concrete explanations are required to understand the practices, stakeholders, and metrics and tools introduced in the study. Another application of this model is combined with Scrum (Dick et al., 2013). This presents the use of the practices for each iteration and final stage. Process assessment is encouraged to start very early for continuous quantifying and improvement of the software process. This application demonstrates still at a high-level, and so falls short of practical guidelines. It is necessary to clearly describe how sustainability objectives are obtained and whose role it is to determine them (Naumann et al., 2015). Lastly, this study focuses on the energy efficiency of testing only, and therefore other potential metrics for green software process are ignored.

6.1.2. The process measurement metrics, tools and techniques for green software

Feasibility is introduced as one of the directly related metric to the quality model proposed for green and sustainable software in Kern et al. (2013). It includes travel, carbon footprint, energy consumption and waste of resources during software development. Another common metric is *energy efficiency*. It is for software product itself at its execution time, but also used for measuring software process efficiency. Dick et al. (2013) introduced the use of this metric for test driven development (TDD) during the test execution of the continuous integration environment in Agile process. Yet, the authors largely ignored other potential metrics such as energy for heating, ventilation, and air conditioning of the offices and energy consumption of workstations (Dick and Naumann, 2010).

In terms of supporting tools for green software process, we have found five categories: operating systems frameworks, fine-grained green computing, performance monitoring counters and metrics, codes written for energy allocation purposes, and virtualization (Mahmoud and Ahmad, 2013). There is also a number of tools for analyzing the priority of the requirements. Saputri and Lee (2016) presents how to use the tool and the example of expected results for supporting decision-making; yet, it does not address other tools or systems for comprehensively supporting green software process.

To cover the aforementioned gaps, *knowledgebase* is introduced by Dick and Naumann (2010). Knowledge is commonly regarded as the main asset to succeed in green sustainable development including checklists, guidelines and educational material (Abdullah et al., 2015). Knowledge management is important as it facilitates simplifying the process of capturing, creating, distributing and sharing knowledge. Knowledge management by "pull" method enables people to search for the knowledge they need by themselves, rather than "push" which is delivering knowledge to people without prior interaction (Abdullah et al., 2015).

6.2. Energy in models and methods

This section presents what and how software models and methods have been considered and incorporated with green requirements.

6.2.1. Software engineering models

A software model is an abstraction of a software component, which is simplified by modeling. Software modeling is a technique to understand the software and communicate with each other. There are different techniques including software modeling languages, notations and supporting tools commonly used in practice. Entity-relationship (ER), Unified Modeling Language (UML), Object-Role Modeling Language (ORM) and Petri-net are widely used for information modeling.

In the implementation of modeling, the software development team should follow the specified green requirements in the requirements phase. It is necessary to ensure the constructed software models are complete, consistent, and correct enough to serve their intended purpose for the stakeholders. The completeness of a software model refers to the degree to which the model fully meets all the requirements specified in the requirements phase. It must include green requirements.

Green requirements trade-offs and risks should be considered as the requirements often compete with other requirements (Chitchyan et al., 2016). Analyzing the consistency of green software models is essential to identify when the models contain any conflicting requirements, constraints, or component descriptions. Interaction analysis must be included in green software engineering. It is to examine interactions between the software components, systems, or users. Software modeling environments can enable the software engineer to review the interaction design and verify the different parts of the software work together in a efficient way.

6.2.2. Software engineering methods

Software engineering methods provide a systematic approach to developing software. Some methods are specific types used in only some of the development phases such as analysis or design, for example, software modeling or data modeling methods. Some methods provide holistic approaches to support the whole life-cycle of software development. Such methods include different types of development approaches including sequential, iterative and agile methods.

Green software can be sequentially developed from one phase to another (e.g., requirements → design → implementation → testing) (Mahmoud and Ahmad, 2013; Shenoy and Eeratta, 2011). In a sequential approach, it is critical to identify, understand, and agree with the green requirements in a very early stage. Any changes or misunderstanding of these requirements may significantly influence the cost and time of the project. To reduce the risks of changes, model verification and validation needs to be considered to ensure that the artifacts of each phase meet the energy requirements.

The development of green software can also be implemented in an iterative and agile way (Dick et al., 2013; Mahmoud and Ahmad, 2013; Naumann et al., 2015; Dick and Naumann, 2010). For each iteration/increment, sustainability (greenability) review and preview processes must be done. The results need to be documented and presented to the stakeholders (e.g., the software development teams, business, and users). After finishing all iterations, final report release and presentations need to follow. A retrospective meeting is essential for the software development team to learn lessons and improve the degree of greenability of the software.

7. Discussion

In this section, we delve into the identified open challenges and trade-offs, providing insights into the broader context of energy concerns in software engineering. We analyze and interpret the findings from our comprehensive investigation, highlighting the implications for software engineers and researchers.

7.1. Shifting trends in software engineering

Overall, green software engineering has been under-researched compared to hardware engineering. However, as energy concerns for software has increased, it has also received much attention from researchers in recent years (Calero and Piattini, 2015). Specifically, energy metrics for software have become one of the key research topics. This leads to increases of studies on this topic such as introduction of energy metrics, general approaches to benchmark systems, and supporting tools and techniques. In addition, all software engineering areas are considered as an important research topic for green software even though some areas are still insufficiently investigated.

More importantly, AI-infused software is strongly required to be ethical and responsible. Most globally known AI risk frameworks take environmental impacts of AI into account, and in the center, energy concerns exist. For example, the four frameworks among the five frameworks (Assessment List for Trustworthy Artificial Intelligence (EU),[7] Algorithmic Impact Assessment (Canada),[8] NSW AI Assurance Framework (Australia),[9] Microsoft Responsible AI Impact Assessment[10] and NIST AI Risk Management Framework(US))[11] we surveyed directly address environmental harm such as energy issue as a key risk of responsible AI systems. This reflects that energy-efficient software has become an essential requirement in any type of software development.

As we highlighted, green software engineering is a promising research area. Nevertheless, some open challenges and limitations definitely exist and they should be discussed importantly for future research.

7.2. Open challenges in Green software engineering

While significant progress has been made in efforts towards Green Software Engineering, numerous open challenges remain, demanding attention and innovative solutions. This section aims to address RQ2, and explores some of these challenges, highlighting the complexities and opportunities that lie ahead in the pursuit of more sustainable and eco-friendly software ecosystems.

7.2.1. Lack of tools and techniques in the green requirements phase

When identifying requirements, the stakeholders may be involved without specific knowledge or with mismatched skills and knowledge among the participating group. This can be a problem, specifically, when a Delphi approach (e.g., niche-sourcing) is adopted for requirements elicitation (Condori-Fernandez et al., 2019). Green requirements are often not detailed; for example, there is no specific goals for energy usage and practitioners just expect that the developed software is not excessively energy-draining (Manotas et al., 2016). It is important to make green requirements specific and detail. In addition, qualitative values are hard to measure (Calero and Piattini, 2015). There should be clear definitions of green metrics and supporting tools. However, the lack of tools or knowledge base for measuring energy for green software has become a challenge in this area (Manotas et al., 2016; Nagappan and Shihab, 2016; Condori-Fernandez et al., 2019).

7.2.2. Difficulties in measuring the impact of software design on energy

Design patterns are commonly used in practice for better productivity and maintainability (Noureddine and Rajan, 2015). In previous studies some design patterns are examined to understand how the patterns contribute to energy efficiency (Manotas et al., 2016; Shenoy and Eeratta, 2011). The experiments of the studies, however, have not been sufficiently conducted across platforms and applications (Bunse and Stiemer, 2013). There are also many different configurations which should be considered to predict or measure energy consumption of software in the design phase. How to generalize the findings remains an open question.

7.2.3. Test environment of multi-versions of software

There are some key considerations such as software building environment when testing different versions of software to measure energy usage. There can be some issues, for example, changing implicit dependencies, evolving compilers and language specifications, and library compatibility (Bangash et al., 2017). This could be remedied by relevant patches and virtualization or building images of an appropriate operating system (Hindle, 2015). Another possible issue is the environment of the testbed (e.g., temperature–heat and cooling). It may affect the systems and software under test, and therefore it is necessary to run tests of the software versions in the same environment.

7.2.4. Adoption of energy-aware development practices

Numerous methods and tools have been proposed for energy efficient software development. However, there is no "silver bullet" for avoiding energy bugs. Software developers need to be trained in energy-awareness, before they can use the available tools and methods correctly (Manotas et al., 2016; Pang et al., 2015). There is a strong demand for organizational policy to support energy-aware development and avoid adoption challenges amongst development teams (Jagroep et al., 2017).

7.2.5. Need for large scale and rigorous verification of energy testing tools

Research has shown the viability of energy-specific test cases (Palit et al., 2011; Couto et al., 2014; Pereira et al., 2020). Automatic energy test generation (Banerjee et al., 2014; Chowdhury et al., 2019a) and test suite minimization (Jabbarvand et al., 2016) offer efficient strategies for reducing the testing time for energy leaks. However, only a few testing methods are verified with actual energy measurements and many rely on energy estimations. Testing frameworks such as SPELLL (Pereira et al., 2020), although promising, have not yet been validated in large scale studies. Moreover, how much energy the testing itself consumes is not clear, and needs to be investigated.

7.2.6. Cost and time constraints in measuring energy consumption of multiple versions of software

For software organizations which deploy commercial software products, understanding the energy consumption of different releases is important. It, however, requires significant efforts, resources and specialized knowledge. Many software organizations are unable to effectively measure the energy consumption of software due to cost and time constraints. In Jagroep et al. (2016) a software energy profiling method is proposed to compare the software energy consumption of different releases. This study attempts to identify energy consumption differences across releases at a fine-grained process level. It may help software organizations to explicitly address these dynamics and report any improvement or deterioration in energy efficiency with a new release. This experiment, however, is limited to a single application and testbed. It should be further studied to be used in the real-world context beyond the laboratory scale.

[7] https://digital-strategy.ec.europa.eu/en/policies/regulatory-framework-ai

[8] https://www.canada.ca/en/government/system/digital-government/digital-government-innovations/responsible-use-ai/algorithmic-impact-assessment.html

[9] https://www.gtlaw.com.au/knowledge/nsw-government-artificial-intelligence-assurance-framework-what-you-need-know

[10] https://blogs.microsoft.com/wp-content/uploads/prod/sites/5/2022/06/Microsoft-RAI-Impact-Assessment-Guide.pdf

[11] https://www.nist.gov/itl/ai-risk-management-framework

7.2.7. Lack of support for green software process

There is a general lack of practical approaches in software engineering process (Chitchyan et al., 2016). The lack of methodological support makes it difficult for practitioners to develop green software as the typical software development life-cycle contains limited concepts of sustainability and greenability (Dick and Naumann, 2010). Moreover, there is a need for change of mentality in a organization. Adopting green practices such as convincing people and getting them to change their way of thinking can be a key challenge. How to make people truly agree on a shared vision of sustainability and work towards it is also critical. To apply sustainable and green design in software development, green requirements must be either driven or approved by the stakeholders. The poor engagement of stakeholders in developing software, therefore, leaves the organization with no choice but to avoid greenability.

7.2.8. Managing trade-offs

We have found challenges in managing trade-offs where conflicting green requirements clash with other competing demands, requiring careful consideration to reach an acceptable compromise throughout the software development life-cycle.

Green requirement conflicts. Sustainability requirements may compete with other requirements. It is often necessary to sacrifice other requirements for reduced energy usage (Manotas et al., 2016). More importantly, trade-offs between green requirements and others (e.g., time to market, performance) should be carefully considered, documented, and managed across the whole life-cycle of the project (Chitchyan et al., 2016; Calero and Piattini, 2015). To this end, resolving conflict by negotiations is necessary (Meridji and Issa, 2013). It includes a series of activities such as identifying the conflict and the stakeholders and involving them to negotiate an acceptable compromise, tracing the decision back to the customer, and implementing the decision.

Maintainability and greenability. Trade-offs exist between maintainability (refactoring of bad smells/anti-patterns) and greenability. While traditional maintenance improves the maintainability of software which refers to the degree of effectiveness and efficiency of modification, green maintenance considers greenability as the degree of environmental friendliness of software. Software greenability in maintenance phase can be explained as feasibility which refers to how software maintenance follows green/sustainable development (Taina, 2011). A few studies have discussed that the maintainability characteristics may have a strong relationship with software greenability such as software energy efficiency based on the power consumption (Calero and Piattini, 2015). It is argued that more modules for fine-grained modular architecture (modularity) imply excessive communication (message) traffic between objects, which can increase energy consumption (Pérez-Castillo and Piattini, 2014). Meanwhile, highly reusable assets (resuability) are highly likely to be optimized leading to better energy efficiency and high degree of software modification (modifiability) may keep its greenability (Calero and Piattini, 2015).

Performance and energy consumption. Software performance and energy efficiency are not always positively correlated. There can be contradicting goals which could require a trade-off to be made (Jagroep et al., 2016). Specifically, mobile/embedded software is highly affected by the battery power (Hindle, 2016). The software can be newly developed or modified to provide better performance and functionality, which might result in a much shorter battery lifetime and ultimately limit the availability of the software. This pinpoints why multiple versions of software should be tested to understand how the differences/changes influence the energy consumption of the software. It also indicates a need for considerations of some modern techniques such as computation offloading to reduce energy consumption but increase performance (Kwon and Tilevich, 2013).

7.3. Practical implications

In this section, we address RQ3 by delving into the practical implications of energy efficiency in Software Engineering, exploring techniques, strategies, and tools that practitioners can employ to optimize energy usage throughout the software life-cycle.

7.3.1. The one-page Green software engineering

Based on our extensive survey, we propose a *one-page solution* which supports practitioners who want to adopt green software engineering. We first show the holistic framework (Table 4), and then present three use cases for different software development methods: Waterfall, Agile and DevOps (Fig. 5).

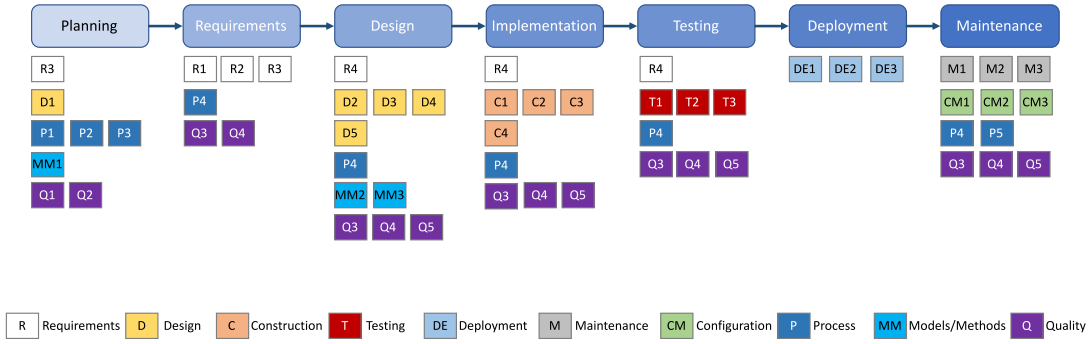
This approach includes critical and fine-grained green tasks and key considerations, which are discussed in the previous sections. It consists of ten software engineering areas (requirements, design, construction, testing, deployment, maintenance, configuration, process, models/methods and quality) to support a broad range of software life-cycle and development methods. In Fig. 5, we demonstrate that how the green tasks/practices can be implemented with Waterfall, Scrum and DevOps methods, respectively. The one-page approach, however, does not limit to those methods presented in the figure. Any software development methods can be considered, and also flexibly tailored as per the development plan, strategies, and circumstance of the software team.

The green tasks can be carried out to align with the software development phases in each method. In the Waterfall method, the tasks should be sequentially implemented, yet measuring green metrics may be taken place in every phase (Fig. 5(a)). Meanwhile, in the Agile method, the development team can repeat most of the green tasks for every sprint (Fig. 5(b)). In this case, the team should take concerns about the limitation of time, budget and the team's capability due to the short period of iterations, and may consider any trade-offs between them. For the DevOps method, our approach can be adopted by distributing the green tasks over both the development and operation phases as shown in Fig. 5(c). The development team and operation team should collaborate to continuously improve and optimize the green process based on the lessons learned, issues and risks, knowledge and feedback from the stakeholders.

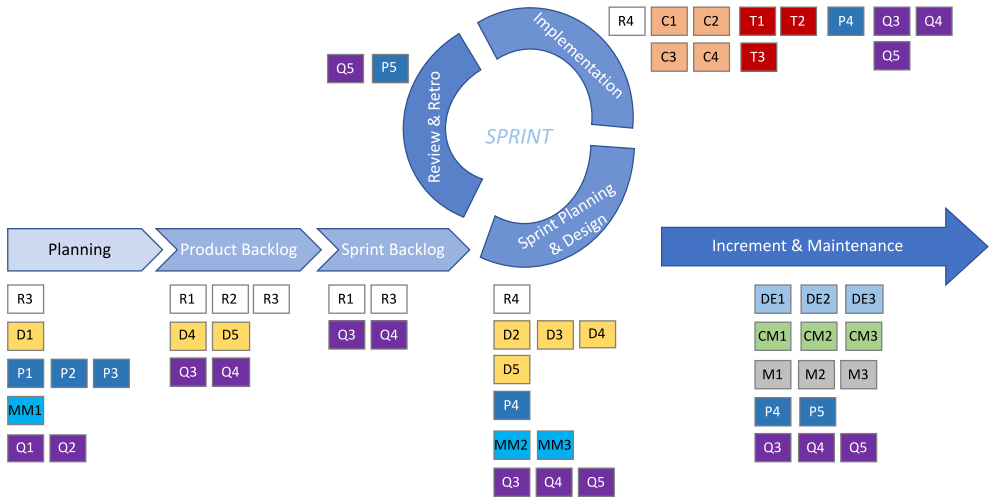
7.3.2. Goals, questions, metrics (GQM) for addressing challenges in software engineering

The challenges identified in this study (Section 7.2) warrant further investigation in future research. Meanwhile, it is important to develop mitigation strategies for the risks associated with these challenges that practitioners may encounter in real-world scenarios. GQM is a comprehensive framework that enables researchers and practitioners to systematically address challenges by setting clear goals, formulating precise questions, and defining relevant metrics (Caldiera and Rombach, 1994). In this section, we propose an example of the GQM framework for green software engineering to tackle the identified challenges.

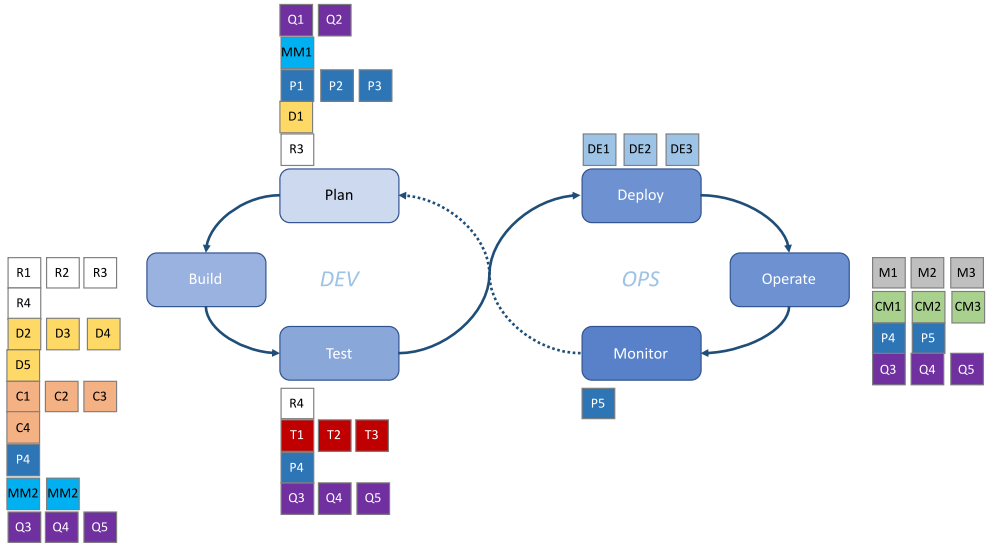
- Goal 1: Improve green requirements elicitation and management (addressing 7.2.1)
  - Question: How can stakeholders with specific knowledge and skills be involved in requirements elicitation for accurate green requirements?
  - Metric 1: Percentage of stakeholders with specialized knowledge and skills involved in the requirements elicitation process.
  - Metric 2: Number of detailed and specific green requirements identified during the elicitation process.
  - Metric 3: Stakeholder satisfaction with the clarity and comprehensiveness of green requirements.



(a) Green Software Engineering for Waterfall.



(b) Green Software Engineering for Agile (Scrum).



(c) Green Software Engineering for DevOps.

Fig. 5. Three use cases of the one-page green software engineering.

Table 4
The one-page green software engineering.

Area	Green task/practice and considerations				
R	R1. Identify product requirements (abstract, implementation, quality).	R2. Identify process requirements (abstract, implementation, quality).	R3. Use of tools and techniques (survey, requirement archetypes).	R4. Assess requirements (by formal guidelines, energy metrics).	
D	D1. Apply green design principles (reusability, less redundancy, etc.).	D2. Use energy-efficient design patterns (e.g., Flyweight, Visitor).	D3. Estimate energy consumption of the selected design patterns.	D4. Use energy-efficient architectural patterns (e.g., MVP).	D5. Estimate energy consumption of selected architectural patterns.
C	C1. Set energy-saving environment.	C2. Consider impact of code obfuscation on energy usage.	C3. Consider impact of data structures/functions on energy usage.	C4. Implement green development strategies (energy dashboard, etc.).	
T	T1. Identify energy test cases for locating energy leaks.	T2. Build automatic test generation (e.g., test selection heuristics).	T3. Minimize a test suit (eliminate redundant test cases).		
DE	DE1. Reduce the size of deployment package using data compression techniques.	DE2. Reduce operational costs (analyze energy patterns, best configuration).	DE3. Choose proper operating models (cloud, fog, edge, offloading, etc.).		
M	M1. Consider refactoring to identify energy-hungry patterns.	M2. Find optimal configuration to consume less power.	M3. Give staff/user training, tutorials for faster maintenance.		
CM	CM1. Measure and compare the energy efficiency of different versions.	CM2. Analyze the energy efficiency of different versions (causes-solutions-decision:accept/reject).	CM3. Estimate energy consumption (use software metrics: CBO, DIT, LOC, etc.).		
P	P1. Understand the key requirements of green software process (sustainability objectives, impacts on environment).	P2. Define process/practices for the entire development life-cycle of software.	P3. Integrate the green process/practices into existing software development process models (waterfall, agile).	P4. Assess the green process/product (green capability, etc.)	P5. Consider knowledge management for sustainable green process for improvement.
MM	MM1. Consider proper software engineering methods and plan green software process.	MM2 Choose a proper modeling language for software components (e.g., ER, UML).	MM3. Assess the quality of a software model (meet green requirements, no conflicting requirements).		
Q	Q1. Define metrics (product and process metrics).	Q2. Adopt approaches to use the metrics.	Q3. Measure metrics (e.g., memory usage, execution time, travel, carbon footprints)	Q4. Consider different options (e.g., base/overhead energy consumption, benchmark metrics).	Q5. Collect energy data and use for judgment on the quality of software.

R:requirements, D:Design, C:Construction, T:Testing, DE:Deployment, M:Maintenance, CM:Configuration Management, P:Development Process, MM:Models/Methods, Q:Quality

- Goal 2: Enhance measurement of software design impact on energy (addressing 7.2.2)
  - Question: How do different design patterns contribute to energy efficiency across platforms and applications?
  - Metric 1: Energy consumption reduction achieved through the implementation of energy-efficient design patterns.
  - Metric 2: Number of configurations considered to predict or measure energy consumption during the design phase.
  - Metric 3: Validity of design patterns' energy efficiency findings across diverse platforms and applications.
- Goal 3: Optimize test environment for energy consumption measurement (addressing 7.2.3)
  - Question: What are the key considerations when testing different versions of software to accurately measure energy usage?
  - Metric 1: Number of software building environment modifications required for energy consumption measurement.
  - Metric 2: Testbed stability and consistency achieved during energy testing of different software versions.
  - Metric 3: Successful replication of energy consumption measurements across different versions of software in the same test environment.
- Goal 4: Foster adoption of energy-aware development practices (addressing 7.2.4)
  - Question: How can software developers be trained in energy-awareness to effectively utilize energy-efficient development tools and methods?
  - Metric 1: Percentage of software developers trained in energy-aware development practices.
  - Metric 2: Adoption rate of energy-efficient development tools and techniques by software development teams.
  - Metric 3: Organizational feedback on the effectiveness of energy-aware development training programs.
- Goal 5: Validate and improve energy testing tools (addressing 7.2.5)

  - Question: How can energy-specific test cases and test suite minimization methods be validated and improved?
  - Metric 1: Number of energy-specific test cases validated with actual energy measurements.
  - Metric 2: Efficiency improvement achieved through automated energy test generation and test suite minimization techniques.
  - Metric 3: Feedback from large-scale studies on the reliability and effectiveness of energy testing frameworks.

- Goal 6: Overcome cost/time constraints in measuring energy consumption (addressing 7.2.6)
  - Question: What strategies can be implemented to measure energy consumption of multiple software versions effectively?
  - Metric 1: Resource utilization (cost and time) for measuring energy consumption across different software releases.
  - Metric 2: Accuracy and reliability of energy consumption measurements across multiple software versions.
  - Metric 3: Improvement or deterioration in energy efficiency reported with each new software release.

- Goal 7: Integrate sustainability into the software engineering process (addressing 7.2.7)
  - Question: How can sustainability and greenability be effectively incorporated into the software engineering process?
  - Metric 1: Number of practical approaches and methodologies developed for integrating sustainability into the software engineering process.
  - Metric 2: Organization-wide adoption of green software development practices and policies.
  - Metric 3: Level of alignment between the software engineering process and sustainability objectives.

- Goal 8: Manage trade-offs in green software engineering (addressing 7.2.8)

  . Goal 8-1: Manage trade-offs in green software engineering
    - Question: How can conflicts between sustainability requirements and other project requirements be resolved?
    - Metric 1: Number of documented trade-offs between green requirements and other project requirements.
    - Metric 2: Stakeholder satisfaction with the negotiated compromises and decisions.
    - Metric 3: Effectiveness of implementing trade-off decisions in the software development life-cycle.

  . Goal 8-2: Balance maintainability and greenability
    - Question: How can maintainability and greenability be balanced in software maintenance activities?
    - Metric 1: Number of refactored bad smells/anti-patterns to improve maintainability.
    - Metric 2: Greenability rating of software maintenance activities based on environmental friendliness criteria.
    - Metric 3: Relationship between maintainability characteristics and software energy efficiency.

  . Goal 8-3: Optimize performance-energy trade-offs
    - Question: How can performance and energy consumption trade-offs be optimized in software design and development?
    - Metric 1: Performance improvement achieved through software modifications.
    - Metric 2: Energy consumption reduction achieved while maintaining acceptable performance levels.
    - Metric 3: Battery lifetime impact of software modifications (e.g.,mobile/embedded systems).

7.3.3. Estimating the energy consumption of AI systems in the supply chain

The increasing concern for responsible AI has brought attention to the need to consider the negative impacts on both humans and the environment. Energy usage of AI models and systems has emerged as a critical factor in promoting responsible AI practices. Recognizing the significance of environmental impact and sustainability, prominent industry frameworks such as NIST AI risk framework and the EU AI risk framework emphasize the assessment and documentation of the environmental implications associated with AI model training and management activities.

Responsible AI extends its scope to encompass the responsible management of supply chains, with the objective of reducing the negative impacts stemming from AI deployments. While Software Bill of Materials (SBOM) has proven valuable in enhancing security and risk management in software supply chains, there remains limited exploration of its application in managing the environmental impact of AI systems (Xia et al., 2023). In parallel, the concept of Green Bill of Materials (GBOM) has primarily been proposed in the manufacturing field to address energy-related issues (Ryu, 2011). There are a few studies on GBOM for software engineering (Gong and Chen, 2009), however, its potential within the software engineering domain, particularly in the context of AI, remains largely unexplored.

AI models heavily rely on external models and dependencies, which can obscure crucial information concerning energy consumption. This lack of transparency makes it challenging to measure and manage the environmental impact effectively. For instance, the utilization of foundation models, such as large language models (LLMs), often conceals energy consumption information. Addressing this opacity and gaining a comprehensive understanding of the energy implications of AI models and systems is crucial for effective energy management and responsible AI practices.

Incorporating with SBOM, Data sheets for datasets (Gebru et al., 2021) and AI model card (Mitchell et al., 2019) or independently, the GBOM can play a critical role in managing and estimating energy consumption within the AI supply chain. By providing a comprehensive list of energy-consuming components and materials used in AI systems, the GBOM enables organizations to assess and monitor the energy efficiency and environmental impact of their AI deployments. It fosters transparency, facilitates informed decision-making, and promotes the adoption of environmentally friendly AI practices.

Utilizing the results of our investigation, the potential elements of GBOM for AI can encompass not only the energy consumption of AI models but also fine-grained information. Table 5 shows the potential elements of GBOM but it is not limited to this.

While the adoption of the GBOM in the software engineering field, specifically within AI, is an area with limited research, its incorporation holds substantial potential for advancing responsible AI practices. By actively exploring and integrating the GBOM into AI frameworks and risk management strategies, organizations can foster sustainable and ethically sound AI implementations.

7.4. Threats to validity

7.4.1. Internal validity

Selection Bias. There is a possibility of selection bias in this study due to the selection of source databases and articles. To mitigate this bias, we developed a research protocol with predefined search strategies and selection criteria, and the process was shared internally among the research team. However, we acknowledge that despite our efforts, certain studies or perspectives may have been inadvertently overlooked or underrepresented. For example, due to time and resource limitations,

Table 5
Potential GBOM elements and considerations for AI systems.

Area	Practice for GBOM	Example of GBOM element
R	– Identify green requirements for suppliers – Assess if the GBOM meets the requirements	– AI model energy metrics/methods/results – Risks of AI model training/management activities – The energy footprint of the training data
D	– Consider energy efficiency of the architecture	– AI model architecture: lightweight and efficient
C	– Consider the environment of AI systems	– Source code, programming language and compiler – Dependencies (e.g., foundation models) – Data structure (e.g., federated learning)
T	– Understand test environment and the impact	– AI model testing information – Model performance and energy consumption
DE	– Energy-efficient deployment methods	– Compression techniques (size of AI model/data) – Operational options (e.g., API, local model)
M	– Consider energy-efficient configuration	– AI model configuration including the parameters, choices, settings, and preferences
	– Faster maintenance to save time and effort	– Staff information for maintenance – Training and education resources/links
CM	– Track the energy consumption of diverse AI system releases	– AI model/data changes, version number and energy consumption of each version
P	– Identify green process/practices for suppliers	– AI model development, training process – Impact analysis results of the process
MM	– Understand AI systems/easy communication	– Development models and methods and artifacts: AI model, application, algorithm, data, and benchmark
Q	– Optimize energy consumption of AI systems	– Metrics/measurement for AI model/data/process – Energy monitoring and reporting

R:requirements, D:Design, C:Construction, T:Testing, DE:Deployment, M:Maintenance, CM:Configuration Management, P:Development Process, MM:Models/Methods, Q:Quality

we had to make some decisions regarding the inclusion of certain databases. Although we included prominent databases such as IEEE Xplore and ACM Digital Library, we excluded others such as ScienceDirect and Scopus. Additionally, to manage the overwhelming volume of search results, we limited the scope of screening to the first relevant 2,000 articles after the initial search. While these decisions were made to ensure a manageable and focused review process, they may have resulted in the omission of relevant studies from these excluded sources.

Time Constraints. The findings of this study are based on the available literature and information up until a specific point in time. Although we aimed to include a broad range of studies, our primary focus was on the period between 2010 and 2021, spanning 10 years. However, it is worth noting that the field of green software engineering is dynamic and constantly evolving. Therefore, new studies, approaches, and challenges may have emerged after our data collection period, which are not captured in this analysis.

7.4.2. External validity

Generalizability to other software contexts. While we have considered general green software practices and considerations, it is important to acknowledge that our findings and recommendations may not be universally applicable to all software development projects. For example, the effectiveness of our suggested solutions, such as the one-page solution, GBOM for AI, and the GQM for green software engineering, should be further validated by practitioners in different types of software development projects or industries to determine their generalizability.

Comprehensive coverage of emerging trends and technologies. This study has made efforts to incorporate insights and considerations related to AI and energy concerns, we acknowledge that the rapid pace of technological advancements may result in new challenges and practices that were not extensively covered in this analysis. To achieve a more comprehensive understanding of the implications of green software engineering in the context of AI and other emerging technologies, it would be beneficial to complement the literature review with additional research methods such as industry stakeholder engagement through interviews, case studies, and surveys. These methods can provide valuable perspectives and real-world experiences, ensuring that the study captures a broader range of challenges and practices in the ever-evolving landscape of software development.

8. Conclusions and future research directions

We conducted a comprehensive survey on energy concerns in software engineering, considering a broad scope of software development areas, including the entire software life cycle, software development management methods, and processes. This study revealed that energy concerns are taken into account in all phases of software development, including the operation of the software, with particular emphasis on the early stages.

While previous studies have explored various green considerations and practices in the selected areas, we also identified several open challenges. For instance, in the area of requirements, we found two types of green requirements: product and process requirements. Each type consists of three levels of requirements, namely abstract, implementation, and quality. While some studies have focused on tools and techniques for effectively managing green requirements, they noted the lack of adequate tools, techniques, and knowledge base, which poses challenges in green requirement management. Additionally, there are trade-offs between requirements, such as performance and energy consumption, as well as maintainability and greenability.

Although attempts have been made to predict and estimate energy consumption during the early stages of software development, such as design time, we did not find a universal approach that can be generalized (Chowdhury et al., 2019b). Addressing this challenge requires further research to facilitate better decision-making processes. Furthermore, using energy-efficient programming languages is another viable approach for energy-saving development. C/C++ is generally recognized as the most energy-efficient language. However, given the unique characteristics, strategies, and environmental contexts of each software project, various factors such as execution time (Pereira et al., 2017; Abdulsalam et al., 2015; Yuki and Rajopadhye, 2013; Pinto et al., 2014), development platforms (Oliveira et al., 2017), data structures (Pinto et al., 2014, 2016), and types of functions used in the software (Dukovic and Varga, 2015) should be comprehensively considered. Additionally, adopting an energy-aware culture, including associated training, policies, and processes, can significantly support green software development.

Several studies emphasized the importance of testing energy consumption when software changes occur. This includes conducting proper analyses of energy usage changes, identifying causes, proposing solutions, and making informed decisions. However, it is worth noting

that testing energy consumption is often associated with cost issues, and many testing methods rely on energy estimations rather than actual measurements.

The key findings of this study have practical implications. Firstly, the green considerations and practices, including the use of tools and techniques, can be utilized to establish a green software engineering framework that facilitates the easy adoption of a green culture. In Section 7.3.1, we propose the one-page solution, which plays a critical role in initiating and promoting green software engineering practices. Secondly, the identified open challenges point towards future research directions that require further investigation and study. Addressing these challenges will contribute to overcoming barriers beyond laboratory-level scenarios and facilitate the practical implementation of green software engineering. To support this, we propose the GQM for green software engineering in Section 7.3.2, providing examples of goals, questions, and metrics that can effectively manage and address the identified challenges in practice. Lastly, regarding the application of emerging trends and technologies, we discussed how to estimate the energy consumption of AI systems in the supply chain, considering the complex dependencies and often invisible energy consumption. We proposed the use of the Green Building Object Model (GBOM) to estimate, register, and track all energy-relevant artifacts, activities, and processes, including AI models, data, and algorithms. In Section 7.3.3, we suggested potential elements of the GBOM for AI based on the results of our investigation.

This study provides valuable insights but it also has several limitations that need to be considered and addressed in future research. Firstly, the limited data sources may introduce research bias and limit the generalization of our findings. To mitigate this, we plan to expand our search scope and include different types of data sources, such as multiple case studies, practitioner surveys, and interviews. Furthermore, although our investigation covered the entire software development process, we did not extensively address the end-of-life phase, which has received limited attention in prior studies. Therefore, conducting case studies or evaluations to further consolidate energy concerns in software engineering is necessary for a more comprehensive understanding of the topic.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

Abdullah, Rusli, Abdullah, Salfarina, Din, Jamilah, Tee, Mcxin, et al., 2015. A systematic literature review of green software development in collaborative knowledge management environment. Int. J. Adv. Comput. Technol. (IJACT) 9, 136.

Abdulsalam, Sarah, Lakomski, Donna, Gu, Qijun, Jin, Tongdan, Zong, Ziliang, 2014. Program energy efficiency: The impact of language, compiler and implementation choices. In: International Green Computing Conference. IEEE, pp. 1–6.

Abdulsalam, Sarah, Zong, Ziliang, Gu, Qijun, Qiu, Meikang, 2015. Using the greenup, powerup, and speedup metrics to evaluate software energy efficiency. In: 2015 Sixth International Green and Sustainable Computing Conference. IGSC, IEEE, pp. 1–8.

Agarwal, Shalabh, Nath, Asoke, Chowdhury, Dipayan, 2012. Sustainable approaches and good practices in green software engineering. Int. J. Res. Rev. Comput. Sci. 3 (1), 1425.

Aggarwal, Karan, Hindle, Abram, Stroulia, Eleni, 2015. Greenadvisor: A tool for analyzing the impact of software evolution on energy consumption. In: 2015 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 311–320.

Aggarwal, Karan, Zhang, Chenlei, Campbell, Joshua Charles, Hindle, Abram, Stroulia, Eleni, 2014. The power of system call traces: predicting the software energy consumption impact of changes. In: CASCON, Vol. 14. pp. 219–233.

Al-Qamash, Amal, Soliman, Iten, Abulibdeh, Rawan, Saleh, Moutaz, 2018. Cloud, fog, and edge computing: A software engineering perspective. In: 2018 International Conference on Computer and Applications. ICCA, IEEE, pp. 276–284.

Alcott, Blake, 2005. Jevons' paradox. Ecolog. Econ. 54 (1), 9–21.

Alharthi, Ahmed D., Spichkova, Maria, Hamilton, Margaret, 2019. Sustainability requirements for elearning systems: a systematic literature review and analysis. Requir. Eng. 24, 523–543.

Ampatzoglou, Apostolos, Charalampidou, Sofia, Stamelos, Ioannis, 2013. Design pattern alternatives: What to do when a GoF pattern fails. In: Proceedings of the 17th Panhellenic Conference on Informatics. pp. 122–127.

Amsel, Nadine, Ibrahim, Zaid, Malik, Amir, Tomlinson, Bill, 2011. Toward sustainable software engineering: NIER track. In: 2011 33rd International Conference on Software Engineering. ICSE, IEEE, pp. 976–979.

Anthony, Bokolo Jnr, Majid, Mazlina Abdul, 2016. Green IS for sustainable decision making in software management. J. Soft Comput. Decis. Support Syst. 3 (3), 20–34.

Ardito, Luca, Procaccianti, Giuseppe, Torchiano, Marco, Vetro, Antonio, 2015. Understanding green software development: A conceptual framework. IT Prof. 17 (1), 44–50.

Baggen, Robert, Correia, José Pedro, Schill, Katrin, Visser, Joost, 2012. Standardized code quality benchmarking for improving software maintainability. Softw. Qual. J. 20 (2), 287–307.

Banerjee, Abhijeet, Chong, Lee Kee, Chattopadhyay, Sudipta, Roychoudhury, Abhik, 2014. Detecting energy bugs and hotspots in mobile apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 588–598.

Bangash, Abdul Ali, Sahar, Hareem, Beg, Mirza Omer, 2017. A methodology for relating software structure with energy consumption. In: 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation. SCAM, IEEE, pp. 111–120.

Becker, Christoph, Chitchyan, Ruzanna, Duboc, Leticia, Easterbrook, Steve, Penzenstadler, Birgit, Seyff, Norbert, Venters, Colin C, 2015. Sustainability design and software: The karlskrona manifesto. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, pp. 467–476.

Berkhout, Frans, Hertin, Julia, 2001. ImpActs of Information and Communication Technologies on Environmental Sustainability: Speculations and Evidence. Report to the OECD, Brighton, 21.

Bourque, Pierre, Dupuis, Robert, Abran, Alain, Moore, James W, Tripp, Leonard, 1999. The guide to the software engineering body of knowledge. IEEE Softw. 16 (6), 35–44.

Broussely, Michel, 2010. Battery requirements for HEVs, PHEVs, and EVs: an overview. In: Electric and Hybrid Vehicles: Power Sources, Models, Sustainability, Infrastructure and the Market. Elsevier, pp. 305–347.

Brundtland Commission, 1987. Report of the world commission on environment and development: Our common future. In: UN Conference on Environment and Development.

Bunse, Christian, Stiemer, Sebastian, 2013. On the energy consumption of design patterns. Softwaretechnik-Trends: Vol. 33, No. 2.

Caldiera, Victor R. Basili1 Gianluigi, Rombach, H. Dieter, 1994. The goal question metric approach. In: Encyclopedia of Software Engineering. pp. 528–532.

Calero, Coral, Bertoa, Manuel F., Moraga, Maria Ángeles, 2013. Sustainability and quality: Icing on the cake. In: RE4SuSy@ RE, Vol. 995.

Calero, Coral, Mancebo Pavón, Javier, García, Félix, 2021. Does maintainability relate to the energy consumption of software? Softw. Qual. J..

Calero, Coral, Moraga, Maria Ángeles, Bertoa, Manuel F., Duboc, Leticia, 2014. Quality in use and software greenability. In: RE4SuSy@ RE. Citeseer, pp. 28–36.

Calero, Coral, Piattini, Mario, 2015. Green in Software Engineering, Vol. 3. Springer.

Capra, Eugenio, Francalanci, Chiara, Slaughter, Sandra A., 2012. Measuring application software energy efficiency. IT Prof. 14 (2), 54–61.

Chatzigeorgiou, Alexander, Stephanides, George, 2002. Energy metric for software systems. Softw. Qual. J. 10 (4), 355–371.

Chen, Feifei, Grundy, John, Schneider, Jean-Guy, Yang, Yun, He, Qiang, 2015. Stresscloud: A tool for analysing performance and energy consumption of cloud applications. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, pp. 721–724.

Chen, FeiFei, Schneider, Jean-Guy, Yang, Yun, Grundy, John, He, Qiang, 2012. An energy consumption model and analysis tool for cloud computing environments. In: 2012 First International Workshop on Green and Sustainable Software. GREENS, IEEE, pp. 45–50.

Chitchyan, Ruzanna, Becker, Christoph, Betz, Stefanie, Duboc, Leticia, Penzenstadler, Birgit, Seyff, Norbert, Venters, Colin C., 2016. Sustainability design in requirements engineering: state of practice. In: Proceedings of the 38th International Conference on Software Engineering Companion. pp. 533–542.

Chowdhury, Shaiful, Borle, Stephanie, Romansky, Stephen, Hindle, Abram, 2019a. GreenScaler: training software energy models with automatic test generation. Empir. Softw. Eng. 24 (4), 1649–1692.

Chowdhury, Shaiful Alam, Hindle, Abram, Kazman, Rick, Shuto, Takumi, Matsui, Ken, Kamei, Yasutaka, 2019b. Greenbundle: An empirical study on the energy impact of bundled processing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 1107–1118.

Condori-Fernandez, Nelly, Lago, Patricia, Luaces, Miguel, Catala, Alejandro, 2019. A nichesourcing framework applied to software sustainability requirements. In: 2019 13th International Conference on Research Challenges in Information Science. RCIS, IEEE, pp. 1–6.

Corbalan, Leonardo, Fernandez, Juan, Cuitiño, Alfonso, Delia, Lisandro, Cáseres, Germán, Thomas, Pablo, Pesado, Patricia, 2018. Development frameworks for mobile devices: A comparative study about energy consumption. In: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems. MOBILESoft '18, Association for Computing Machinery, New York, NY, USA, pp. 191—201.

Couto, Marco, Carção, Tiago, Cunha, Jácome, Fernandes, João Paulo, Saraiva, João, 2014. Detecting anomalous energy consumption in android applications. In: Brazilian Symposium on Programming Languages. Springer, pp. 77–91.

Couto, Marco, Pereira, Rui, Ribeiro, Francisco, Rua, Rui, Saraiva, João, 2017. Towards a green ranking for programming languages. In: Proceedings of the 21st Brazilian Symposium on Programming Languages. pp. 1–8.

Cruz, Luis, Abreu, Rui, 2019. Catalog of energy patterns for mobile applications. Empir. Softw. Eng. 24 (4), 2209–2235.

Dick, Markus, Drangmeister, Jakob, Kern, Eva, Naumann, Stefan, 2013. Green software engineering with agile methods. In: 2013 2nd International Workshop on Green and Sustainable Software. GREENS, IEEE, pp. 78–85.

Dick, Markus, Naumann, Stefan, 2010. Enhancing software engineering processes towards sustainable software product design. In: EnviroInfo. Citeseer, pp. 706–715.

Dick, Markus, Naumann, Stefan, Kuhn, Norbert, 2010. A model and selected instances of green and sustainable software. In: What Kind of Information Society? Governance, Virtuality, Surveillance, Sustainability, Resilience: 9th IFIP TC 9 International Conference, HCC9 2010 and 1st IFIP TC 11 International Conference, CIP 2010, Held As Part of WCC 2010, Brisbane, Australia, September 20-23, 2010. Proceedings. Springer, pp. 248–259.

Dukovic, Marko, Varga, Ervin, 2015. Load profile-based efficiency metrics for code obfuscators. Acta Polytech. Hung. 12 (5).

Erdelyi, Krisztina, 2013. Special factors of development of green software supporting eco sustainability. In: 2013 IEEE 11th International Symposium on Intelligent Systems and Informatics. SISY, IEEE, pp. 337–340.

Feitosa, Daniel, Alders, Rutger, Ampatzoglou, Apostolos, Avgeriou, Paris, Nakagawa, Elisa Yumi, 2017. Investigating the effect of design patterns on energy consumption. J. Softw. Evol. Process 29 (2), e1851.

Ferreira, Denzil, Dey, Anind K., Kostakos, Vassilis, 2011. Understanding human-smartphone concerns: a study of battery life. In: International Conference on Pervasive Computing. Springer, pp. 19–33.

Fowler, Martin, 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.

Gebru, Timnit, Morgenstern, Jamie, Vecchione, Briana, Vaughan, Jennifer Wortman, Wallach, Hanna, Iii, Hal Daumé, Crawford, Kate, 2021. Datasheets for datasets. Commun. ACM 64 (12), 86–92.

Georges, Andy, Buytaert, Dries, Eeckhout, Lieven, 2007. Statistically rigorous java performance evaluation. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications. OOPSLA '07, Association for Computing Machinery, New York, NY, USA, pp. 57–76.

Georgiou, Stefanos, Rizou, Stamatia, Spinellis, Diomidis, 2019a. Software development lifecycle for energy efficiency: Techniques and tools. ACM Comput. Surv. 52 (4), 81.

Georgiou, Stefanos, Rizou, Stamatia, Spinellis, Diomidis, 2019b. Software development lifecycle for energy efficiency: techniques and tools. ACM Comput. Surv. 52 (4), 1–33.

Gong, Dah-Chuan, Chen, Jia-Ling, 2009. Developing a software system to manage green products. In: 2009 International Conference on New Trends in Information and Service Science. IEEE, pp. 1076–1080.

Goodland, Robert, et al., 2002. Sustainability: human, social, economic and environmental. In: Encyclopedia of Global Environmental Change, Vol. 5. Wiley and Sons, pp. 481–491.

Gottschalk, Marion, Josefiok, Mirco, Jelschen, Jan, Winter, Andreas, 2012. Removing energy code smells with reengineering services. In: Goltz, Ursula, Magnor, Marcus, Appelrath, Hans-Jürgen, Matthies, Herbert K., Balke, Wolf-Tilo, Wolf, Lars (Eds.), INFORMATIK 2012. Gesellschaft für Informatik e.V., Bonn, pp. 441–455.

Hao, Shuai, Li, Ding, Halfond, William G.J., Govindan, Ramesh, 2013. Estimating mobile application energy consumption using program analysis. In: 2013 35th International Conference on Software Engineering. ICSE, IEEE, pp. 92–101.

Hasan, Khalid, Biswas, Kamanashis, Ahmed, Khandakar, Nafi, Nazmus S, Islam, Md Saiful, 2019. A comprehensive review of wireless body area network. J. Netw. Comput. Appl. 143, 178–198.

Hasan, Samir, King, Zachary, Hafiz, Munawar, Sayagh, Mohammed, Adams, Bram, Hindle, Abram, 2016. Energy profiles of java collections classes. In: Proceedings of the 38th International Conference on Software Engineering. ICSE '16, Association for Computing Machinery, New York, NY, USA, pp. 225–236.

Hilty, Lorenz, Lohmann, Wolfgang, Huang, Elaine M., 2011. Sustainability and ICT-an overview of the field. Notizie di POLITEIA 27 (104), 13–28.

Hindle, Abram, 2015. Green mining: a methodology of relating software change and configuration to power consumption. Empir. Softw. Eng. 20 (2), 374–409.

Hindle, Abram, 2016. Green software engineering: the curse of methodology. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, Vol. 5. SANER, IEEE, pp. 46–55.

Hogan, Trish, 2009. Overview of TPC benchmark e: The next generation of OLTP benchmarks. In: Technology Conference on Performance Evaluation and Benchmarking. Springer, pp. 84–98.

Hönig, Timo, Eibel, Christopher, Kapitza, Rüdiger, Schröder-Preikschat, Wolfgang, 2012. SEEP: exploiting symbolic execution for energy-aware programming. Oper. Syst. Rev. 45 (3), 58–62.

Hönig, Timo, Janker, Heiko, Eibel, Christopher, Mihelic, Oliver, Kapitza, Rüdiger, 2014. Proactive energy-aware programming with PEEK. In: 2014 Conference on Timely Results in Operating Systems. TRIOS 14, USENIX Association, Broomfield, CO.

Hsu, Hwa-You, Orso, Alessandro, 2009. MINTS: A general framework and tool for supporting test-suite minimization. In: 2009 IEEE 31st International Conference on Software Engineering. IEEE, pp. 419–429.

International Energy Agency, 2021. Global energy review 2021. https://www.iea.org/news/global-carbon-dioxide-emissions-are-set-for-their-second-biggest-increase-in-history.

Jabbarvand, Reyhaneh, Sadeghi, Alireza, Bagheri, Hamid, Malek, Sam, 2016. Energy-aware test-suite minimization for android apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. pp. 425–436.

Jagroep, E., Broekman, J., Werf, J. M. E. M. van der, Lago, P., Brinkkemper, S., Blom, L., Vliet, R. van, 2017. Awakening awareness on energy consumption in software engineering. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Society Track. ICSE-SEIS, pp. 76–85.

Jagroep, Erik A., van der Werf, Jan Martijn, Brinkkemper, Sjaak, Procaccianti, Giuseppe, Lago, Patricia, Blom, Leen, van Vliet, Rob, 2016. Software energy profiling: Comparing releases of a software product. In: Proceedings of the 38th International Conference on Software Engineering Companion. pp. 523–532.

Johann, Timo, Dick, Markus, Kern, Eva, Naumann, Stefan, 2011. Sustainable development, sustainable software, and sustainable software engineering: an integrated approach. In: 2011 International Symposium on Humanities, Science and Engineering Research. IEEE, pp. 34–39.

Kansal, Aman, Zhao, Feng, Liu, Jie, Kothari, Nupur, Bhattacharya, Arka A., 2010. Virtual machine power metering and provisioning. In: Proceedings of the 1st ACM Symposium on Cloud Computing. pp. 39–50.

Kazman, Rick, Haziyev, Serge, Yakuba, Andriy, Tamburri, Damian A, 2018. Managing energy consumption as an architectural quality attribute. IEEE Softw. 35 (5), 102–107.

Kern, Eva, Dick, Markus, Naumann, Stefan, Guldner, Achim, Johann, Timo, 2013. Green software and green software engineering–definitions, measurements, and quality aspects. In: First International Conference on Information and Communication Technologies for Sustainability. ICT4S2013, 2013b ETH Zurich, pp. 87–91.

Kiehne, Heinz Albert, 2003. Battery Technology Handbook, Vol. 118. CRC Press.

Kipp, Alexander, Jiang, Tao, Fugini, Mariagrazia, 2011. Green metrics for energy-aware IT systems. In: 2011 International Conference on Complex, Intelligent, and Software Intensive Systems. IEEE, pp. 241–248.

Kitchenham, Barbara, Brereton, O Pearl, Budgen, David, Turner, Mark, Bailey, John, Linkman, Stephen, 2009. Systematic literature reviews in software engineering–a systematic literature review. Inf. Softw. Technol. 51 (1), 7–15.

Kounev, Samuel, Lange, Klaus-Dieter, von Kistowski, Jóakim, 2020. Storage benchmarks. In: Systems Benchmarking. Springer, pp. 285–300.

Kwon, Young-Woo, Tilevich, Eli, 2013. Reducing the energy consumption of mobile applications behind the scenes. In: 2013 IEEE International Conference on Software Maintenance. IEEE, pp. 170–179.

Lago, Patricia, Kazman, Rick, Meyer, Niklaus, Morisio, Maurizio, Müller, Hausi A, Paulisch, Frances, 2013. Exploring initial challenges for green software engineering: summary of the first GREENS workshop, at ICSE 2012. ACM SIGSOFT Softw. Eng. Notes 38 (1), 31–33.

Lami, Giuseppe, Buglione, Luigi, 2012. Measuring software sustainability from a process-centric perspective. In: 2012 Joint Conference of the 22nd International Workshop on Software Measurement and the 2012 Seventh International Conference on Software Process and Product Measurement. IEEE, pp. 53–59.

Lami, Giuseppe, Fabbrini, Fabrizio, Fusani, Mario, 2012. Software sustainability from a process-centric perspective. In: European Conference on Software Process Improvement. Springer, pp. 97–108.

Lange, K., 2009. Identifying shades of green: The SPECpower benchmarks. IEEE Ann. Hist. Comput. 42 (03), 95–97.

Linares-Vásquez, Mario, Bavota, Gabriele, Cárdenas, Carlos Eduardo Bernal, Oliveto, Rocco, Di Penta, Massimiliano, Poshyvanyk, Denys, 2015. Optimizing energy consumption of guis in android apps: A multi-objective approach. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 143–154.

Mahmoud, Sara S., Ahmad, Imtiaz, 2013. A green model for sustainable software engineering. Int. J. Softw. Eng. Appl. 7 (4), 55–74.

Manotas, Irene, Bird, Christian, Zhang, Rui, Shepherd, David, Jaspan, Ciera, Sadowski, Caitlin, Pollock, Lori, Clause, James, 2016. An empirical study of practitioners' perspectives on green software engineering. In: 2016 IEEE/ACM 38th International Conference on Software Engineering. ICSE, IEEE, pp. 237–248.

Manotas, Irene, Pollock, Lori, Clause, James, 2014. Seeds: A software engineer's energy-optimization decision support framework. In: Proceedings of the 36th International Conference on Software Engineering. pp. 503–514.

Meridji, Kenza, Issa, Ghassan, 2013. A development approach of software requirements for renewable energy applications using fundamental principles of software engineering. In: 2013 1st International Conference & Exhibition on the Applications of Information Technology To Renewable Energy Processes and Systems. IEEE, pp. 107–112.

Mitchell, Margaret, Wu, Simone, Zaldivar, Andrew, Barnes, Parker, Vasserman, Lucy, Hutchinson, Ben, Spitzer, Elena, Raji, Inioluwa Deborah, Gebru, Timnit, 2019. Model cards for model reporting. In: Proceedings of the Conference on Fairness, Accountability, and Transparency. pp. 220–229.

Morales, Rodrigo, Saborido, Rubén, Khomh, Foutse, Chicano, Francisco, Antoniol, Giuliano, 2017. Earmo: an energy-aware refactoring approach for mobile apps. IEEE Trans. Softw. Eng. 44 (12), 1176–1206.

Nagappan, Meiyappan, Shihab, Emad, 2016. Future trends in software engineering research for mobile apps. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, Vol. 5. SANER, IEEE, pp. 21–32.

Naumann, Stefan, Dick, Markus, Kern, Eva, Johann, Timo, 2011. The greensoft model: A reference model for green and sustainable software and its engineering. Sustain. Comput. Inf. Syst. 1 (4), 294–304.

Naumann, Stefan, Kern, Eva, Dick, Markus, Johann, Timo, 2015. Sustainable software engineering: Process and quality models, life cycle, and social aspects. In: ICT Innovations for Sustainability. Springer, pp. 191–205.

Noureddine, Adel, Rajan, Ajitha, 2015. Optimising energy consumption of design patterns. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, pp. 623–626.

Oliveira, Wellington, Oliveira, Renato, Castor, Fernando, 2017. A study on the energy consumption of android app development approaches. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories. MSR, pp. 42–52.

Oliveira, Wellington, Oliveira, Renato, Castor, Fernando, Pinto, Gustavo, Fernandes, João Paulo, 2021. Improving energy-efficiency by recommending Java collections. Empir. Softw. Eng. 26 (3), 1–45.

Palit, Rajesh, Arya, Renuka, Naik, Kshirasagar, Singh, Ajit, 2011. Selection and execution of user level test cases for energy cost evaluation of smartphones. In: Proceedings of the 6th International Workshop on Automation of Software Test. pp. 84–90.

Pang, Candy, Hindle, Abram, Adams, Bram, Hassan, Ahmed E., 2015. What do programmers know about software energy consumption? IEEE Softw. 33 (3), 83–89.

Park, Jae Jin, Hong, Jang-Eui, Lee, Sang-Ho, 2014. Investigation for software power consumption of code refactoring techniques. In: SEKE. pp. 717–722.

Pathak, Abhinav, Hu, Y. Charlie, Zhang, Ming, 2012. Where is the energy spent inside my app? Fine grained energy accounting on smartphones with eprof. In: Proceedings of the 7th ACM European Conference on Computer Systems. pp. 29–42.

Paul, Kolin, Kundu, Tapas Kumar, 2010. Android on mobile devices: An energy perspective. In: 2010 10th IEEE International Conference on Computer and Information Technology. pp. 2421–2426.

Penzenstadler, Birgit, 2012. Supporting sustainability aspects in software engineering. In: 3rd International Conference on Computational Sustainability. CompSust.

Penzenstadler, Birgit, 2013. Towards a definition of sustainability in and for software engineering. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. pp. 1183–1185.

Penzenstadler, Birgit, Femmer, Henning, 2013. A generic model for sustainability with process-and product-specific instances. In: Proceedings of the 2013 Workshop on Green in/By Software Engineering. pp. 3–8.

Penzenstadler, Birgit, Raturi, Ankita, Richardson, Debra, Calero, Coral, Femmer, Henning, Franch, Xavier, 2014a. Systematic mapping study on software engineering for sustainability (SE4s). In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. pp. 1–14.

Penzenstadler, Birgit, Raturi, Ankita, Richardson, Debra, Tomlinson, Bill, 2014b. Safety, security, now sustainability: The nonfunctional requirement for the 21st century. IEEE Softw. 31 (3), 40–47.

Pereira, Rui, Carção, Tiago, Couto, Marco, Cunha, Jácome, Fernandes, João Paulo, Saraiva, João, 2020. SPELLing out energy leaks: Aiding developers locate energy inefficient code. J. Syst. Softw. 161, 110463.

Pereira, Rui, Couto, Marco, Ribeiro, Francisco, Rua, Rui, Cunha, Jácome, Fernandes, João Paulo, Saraiva, João, 2017. Energy efficiency across programming languages: how do energy, time, and memory relate? In: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering. pp. 256–267.

Pérez-Castillo, Ricardo, Piattini, Mario, 2014. Analyzing the harmful effect of god class refactoring on power consumption. IEEE Softw. 31 (3), 48–54.

Pinto, Gustavo, Castor, Fernando, 2017. Energy efficiency: a new concern for application software developers. Commun. ACM 60 (12), 68–75.

Pinto, Gustavo, Castor, Fernando, Liu, Yu David, 2014. Understanding energy behaviors of thread management constructs. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. OOPSLA '14, Association for Computing Machinery, New York, NY, USA, pp. 345–360.

Pinto, G., Liu, K., Castor, F., Liu, Y.D., 2016. A comprehensive study on the energy efficiency of java's thread-safe collections. In: 2016 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 20–31.

Poess, Meikel, Nambiar, Raghunath Othayoth, Vaid, Kushagra, Stephens, Jr., John M, Huppler, Karl, Haines, Evan, 2010. Energy benchmarks: a detailed analysis. In: Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking. pp. 131–140.

Raj, Athul, Jithish, J., Sankaran, Sriram, 2017. Modelling the impact of code obfuscation on energy usage. In: DIAS/EDUDM@ ISEC.

Ramachandran, Gowri Sankar, Daniels, Wilfried, Matthys, Nelson, Huygens, Christophe, Michiels, Sam, Joosen, Wouter, Meneghello, James, Lee, Kevin, Cañete, Eduardo, Rodriguez, Manuel Diaz, Hughes, Danny, 2015. Measuring and modeling the energy cost of reconfiguration in sensor networks. IEEE Sens. J. 15 (6), 3381–3389.

Roher, Kristin, Richardson, Debra, 2013. A proposed recommender system for eliciting software sustainability requirements. In: 2013 2nd International Workshop on User Evaluations for Software Engineering Researchers. USER, IEEE, pp. 16–19.

Ryu, K., 2011. Green product and production information management in the fractal manufacturing system. In: Proc. ICPR21. Stuttgart, Germany, pp. 1–8.

Sahar, Hareem, Bangash, Abdul A., Beg, Mirza O., 2019. Towards energy aware object-oriented development of android applications. Sustain. Comput. Inf. Syst. 21, 28–46.

Sahin, Cagri, Cayci, Furkan, Gutiérrez, Irene Lizeth Manotas, Clause, James, Kiamilev, Fouad, Pollock, Lori, Winbladh, Kristina, 2012. Initial explorations on design pattern energy usage. In: 2012 First International Workshop on Green and Sustainable Software. GREENS, IEEE, pp. 55–61.

Sahin, Cagri, Pollock, Lori, Clause, James, 2014. How do code refactorings affect energy usage? In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 1–10.

Sahin, Cagri, Wan, Mian, Tornquist, Philip, McKenna, Ryan, Pearson, Zachary, Halfond, William G.J., Clause, James, 2016. How does code obfuscation impact energy usage? J. Softw. Evol. Process 28 (7), 565–588.

Saputri, Theresia Ratih Dewi, Lee, Seok-Won, 2016. Incorporating sustainability design in requirements engineering process: A preliminary study. In: Asia Pacific Requirements Engineering Conference. Springer, pp. 53–67.

Saputri, Theresia Ratih Dewi, Lee, Seok-Won, 2021. Integrated framework for incorporating sustainability design in software engineering life-cycle: An empirical study. Inf. Softw. Technol. 129, 106407.

Seo, Chiyoung, Malek, Sam, Medvidovic, Nenad, 2008. Estimating the energy consumption in pervasive java-based systems. In: 2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications. PerCom, IEEE, pp. 243–247.

Shenoy, Sanath S., Eeratta, Raghavendra, 2011. Green software development model: An approach towards sustainable software development. In: 2011 Annual IEEE India Conference. IEEE, pp. 1–6.

Siegmund, Norbert, Rosenmüller, Marko, Apel, Sven, 2010. Automating energy optimization with features. In: Proceedings of the 2nd International Workshop on Feature-Oriented Software Development. pp. 2–9.

Singh, Digvijay, Peterson, Peter AH, Reiher, Peter L, Kaiser, William J, 2010. The Atom LEAP Platform for Energy-Efficient Embedded Computing: Architecture, Operation, and System Implementation. UCLA Technical Report.

Steigerwald, Bob, Chabukswar, Rajshree, Krishnan, Karthik, Vega, J.D., 2008. Creating Energy Efficient Software. Intel White Paper.

Taina, Juha, 2011. Good, bad, and beautiful software-in search of green software quality factors. Cepis Upgrade 12 (4), 22–27.

Tanelli, Mara, Ardagna, Danilo, Lovera, Marco, Zhang, Li, 2008. Model identification for energy-aware management of web service systems. In: International Conference on Service-Oriented Computing. Springer, pp. 599–606.

Van Loon, Han, 2004. Process Assessment and ISO/IEC 15504: A Reference Book, Vol. 775. Springer Science & Business Media.

Venters, Colin C., Capilla, Rafael, Betz, Stefanie, Penzenstadler, Birgit, Crick, Tom, Crouch, Steve, Nakagawa, Elisa Yumi, Becker, Christoph, Carrillo, Carlos, 2018. Software sustainability: Research and practice from a software architecture viewpoint. J. Syst. Softw. 138, 174–188.

Venters, Colin C., Capilla, Rafael, Nakagawa, Elisa Yumi, Betz, Stefanie, Penzenstadler, Birgit, Crick, Tom, Brooks, Ian, 2023. Sustainable software engineering: Reflections on advances in research and practice. Inf. Softw. Technol. 107316.

Venters, Colin C., Kocak, Sedef Akinli, Betz, Stefanie, Brooks, Ian, Capilla, Rafael, Chitchyan, Ruzanna, Duboc, Letícia, Heldal, Rogardt, Moreira, Ana, Oyedeji, Shola, et al., 2021. Software sustainability: beyond the tower of babel. In: 2021 IEEE/ACM International Workshop on Body of Knowledge for Software Sustainability. BoKSS, IEEE, pp. 3–4.

Vetro, Antonio, Ardito, Luca, Procaccianti, Giuseppe, Morisio, Maurizio, et al., 2013. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. In: Proceedings of ENERGY 2013: The Third International Conference on Smart Grids, Green Communications and IT Energy-Aware Technologies. pp. 34–39.

Warade, Mehul, Schneider, Jean-Guy, Lee, Kevin, 2021. FEPAC: A framework for evaluating parallel algorithms on cluster architectures. In: 2021 Australasian Computer Science Week Multiconference. ACSW '21, Association for Computing Machinery, New York, NY, USA.

Webster, Jane, Watson, Richard T., 2002. Analyzing the past to prepare for the future: Writing a literature review. MIS Q. xiii–xxiii.

Wilke, Claas, Götz, Sebastian, Richly, Sebastian, 2013. Jouleunit: a generic framework for software energy profiling and testing. In: Proceedings of the 2013 Workshop on Green in/By Software Engineering. pp. 9–14.

Wu, Taiyang, Wu, Fan, Redoute, Jean-Michel, Yuce, Mehmet Rasit, 2017. An autonomous wireless body area network implementation towards IoT connected healthcare applications. IEEE Access 5, 11413–11422.

Xia, Boming, Bi, Tingting, Xing, Zhenchang, Lu, Qinghua, Zhu, Liming, 2023. An empirical study on software bill of materials: Where we stand and the road ahead. arXiv preprint arXiv:2301.05362.

Xing, Jiang, Zhu, Yunru, 2009. A survey on body area network. In: 2009 5th International Conference on Wireless Communications, Networking and Mobile Computing. IEEE, pp. 1–4.

Yuki, Tomofumi, Rajopadhye, Sanjay, 2013. Folklore confirmed: Compiling for speed=compiling for energy. In: International Workshop on Languages and Compilers for Parallel Computing. Springer, pp. 169–184.

Zhang, Chenlei, Hindle, Abram, 2014. A green miner's dataset: mining the impact of software change on energy consumption. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 400–403.

Sung Une Lee is a research scientist at CSIRO's Data61. She is currently working on Software Engineering and Responsible AI. She has over 10 years of industry experience in Software Engineering and Project Management. Her research interests include responsible AI, software engineering, data governance and project management.

Niroshinie Fernando is a senior lecturer in Software Engineering & Industry Practice Lead, SwEng & IoT at Deakin University. She has previous industry experience, always looking for interesting problems to solve. Her interest areas include IoT, mobile computing, edge & cloud, smart cities, human-centered software, and ethical robots.

Kevin Lee is an associate professor at Deakin University. He is a computer science researcher with over 80 International publications. He experiences lecturing in Computer Science, reviewer, and chair of sessions at conferences. His specialties include Internet of Things (IoT), robotics, applied distributed computing, and cloud computing.

Jean-Guy holds a Ph.D. degree in Computer Science and Applied Mathematics from the University of Bern, Switzerland, and has more than 20 years' experience in the Higher Education Sector in Australia. He has held a variety of leadership positions and is currently the Associate Dean (Education) of the Faculty of IT at Monash University. His main research interests lie in the general area of reliable software technologies with a special focus on component technologies and user-centered approaches.