

Green Software Engineering

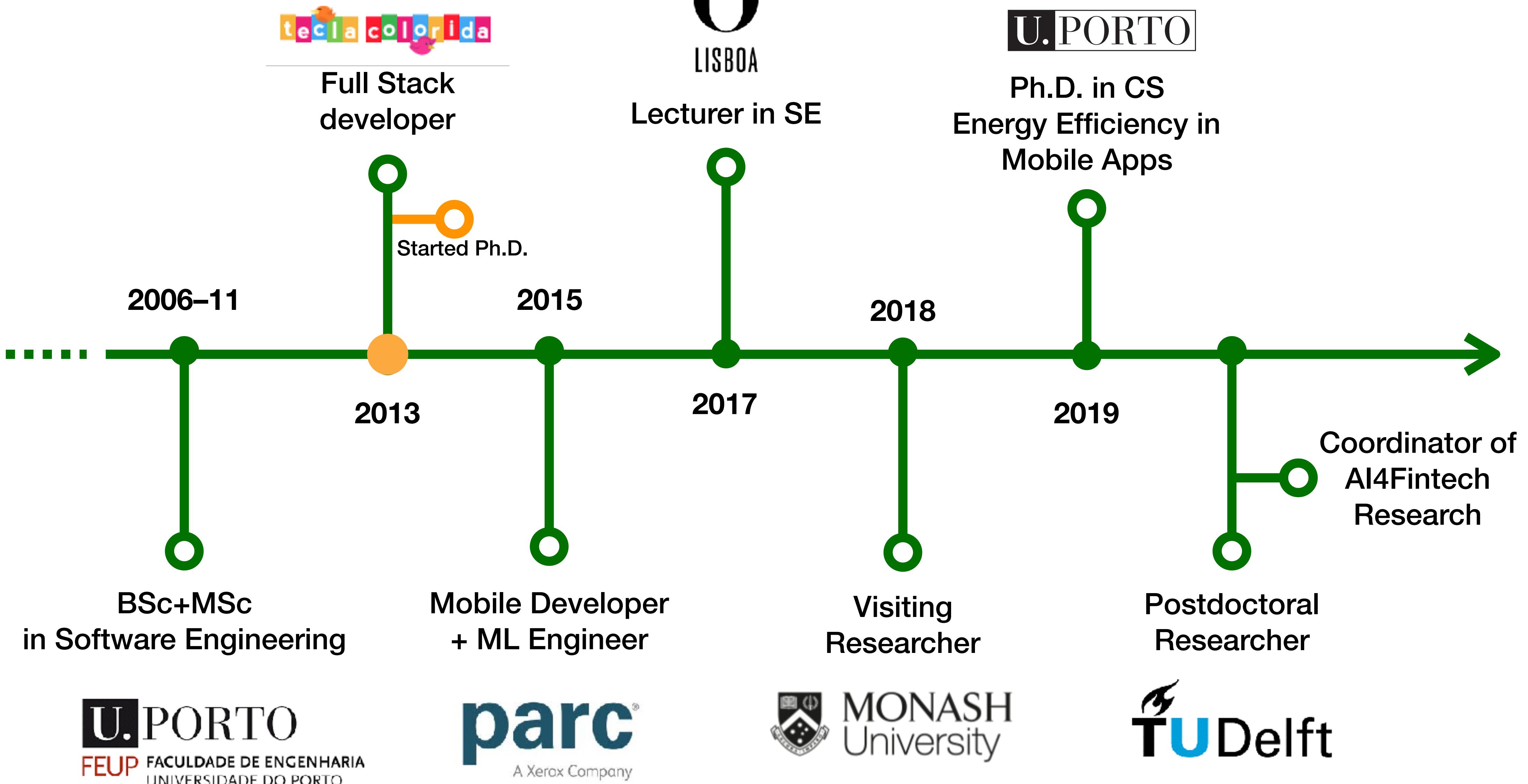
Guest Lecture in Software Architecture (IN4315)

Luís Cruz

 @luismcruz

 L.Cruz@tudelft.nl

 <https://luiscruz.github.io/>



AI for Fintech Research (AFR)

- Collaboration between **ING** and **TU Delft**.
- Artificial Intelligence, Data Analytics, and Software Analytics in the context of **FinTech**.
- Officially started in January 2020.
- 10 research tracks, 5 years.
- Website: <https://se.ewi.tudelft.nl/ai4fintech/>
- Twitter: [@Ai4Fintech](https://twitter.com/Ai4Fintech)



Software Sustainability

Sustainability Design and Software: The Karlskrona Manifesto

Christoph Becker
Faculty of Information
University of Toronto
Toronto, ON, Canada
christoph.becker@utoronto.ca

Ruzanna Chitchyan
Dept of Computer Science
University of Leicester
Leicester, UK
ru256@leicester.ac.uk

Leticia Duboc
Dept of Inf. & Computer Science
State Univ. of Rio de Janeiro
Rio de Janeiro, Brazil
leticia@imc.ufrj.br

Steve Easterbrook
Dept of Computer Science
University of Toronto
Toronto, ON, Canada
smc@cs.toronto.ca

Birgit Penzenstadler
Institute for Software Research
University of California, Irvine
Irvine, California, US
bpenzens@uci.edu

Norbert Seyff
Dept of Informatics
University of Zurich
Zurich, Switzerland
seyff@ifi.uzh.ch

Colin C. Venters
School of Computing & Engineering
University of Huddersfield
Huddersfield, UK
c.venters@hud.ac.uk

Abstract—Sustainability has emerged as a broad concern for society. Many engineering disciplines have been grappling with challenges in how we sustain technical, social and ecological systems. In the software engineering community, for example, maintainability has been a concern for a long time. But too often, these issues are treated in isolation from one another. Misperceptions among practitioners and research communities persist, rooted in a lack of coherent understanding of sustainability, and how it relates to software systems research and practice. This article presents a cross-disciplinary initiative to create a common ground and a point of reference for the global community of research and practice in software and sustainability, to be used for effectively communicating key issues, goals, values and principles of sustainability design for software-intensive systems. The centerpiece of this effort is the *Karlskrona Manifesto for Sustainability Design*, a vehicle for a much needed conversation about sustainability within and beyond the software community, and an articulation of the fundamental principles underpinning design choices that affect sustainability. We describe the motivation for developing this manifesto, including some considerations of the genre of the manifesto as well as the dynamics of its creation. We illustrate the collaborative reflective writing process and present the current edition of the manifesto itself. We assess immediate implications and applications of the articulated principles, compare these to current practice, and suggest future steps.

I. INTRODUCTION

It is clear that society is facing major sustainability challenges that require long-term, joined-up thinking. How do we sustain our technical infrastructures, given how much we rely on them for everything from communication and navigation through to storing health records, identifying security threats, and keeping the lights on? How do we sustain prosperity in society, given the erosion of trust in our political institutions and a growing inequality in ownership of resources? And, above all, how do we sustain the planetary systems that support life on earth, in the face of accumulation of pollutants, species loss, and accelerating climate change?

The discipline of Software Engineering (SE) has a major role to play in sustainability, because of the extent to which software systems mediate so many aspects of our lives. However, software practice has a tendency to focus only on the immediate effects and tangible benefits of software products and platforms. SE research has, for the most part, focused on increasing the reliability, efficiency and cost-benefit relation of software products for their owners, through a focus on processes, methods, models and techniques to create, verify and validate software systems and keep them operational.

The lack of long-term thinking in software engineering research and practice has been critiqued throughout the history of the discipline. For example, software maintenance and evolution were raised as concerns even at the very first software engineering conference [1]. Since then efforts to increase the maintainability of software products and facilitate their evolution have often focused on improving architecture, decreasing lifecycle costs and managing technical debt [2]. Neumann has criticized the lack of long-term thinking over security considerations in SE [3]. For our digital information assets, some now speak of a ‘digital dark age’ [4], where, having discarded analog media in preference for digital, we now find that many of these assets become unreadable, due, in part, to the rapid lifecycles of software technology.

While progress has been made on design for maintainability of software *per se*, considerations that extend beyond immediate software product qualities and user benefits are generally treated as secondary concerns: optional qualities to be addressed only after the system under design has been shown to be a success in terms of technical and/or marketing criteria. The larger impact of software artefacts on society and the natural environment is not routinely analyzed. But by trading off longer-term sustainability questions for shorter-term success criteria, we accumulate threats to sustainability. We argue that this trade-off itself is unnecessary. As Neumann

(Becker, 2015)

Software Sustainability

- Individual. Mental and physical well-being, education, freedom, self-respect, mobility, agency.
 - Social. Social equity, justice, employment, democracy.
 - Technical. Maintenance, innovation, obsolescence, data integrity.
- Economic. Wealth creation, prosperity, profitability, capital investment, income.
 - Environmental. Ecosystems, raw resources, climate change, food production, water, pollution, waste.



Sustainability Design and Software: The Karlskrona Manifesto

Christoph Becker Faculty of Information University of Toronto Toronto, ON, Canada christoph.becker@utoronto.ca	Ruzanna Chitchyan Dept of Computer Science University of Leicester Leicester, UK rc256@leicester.ac.uk	Lelicia Dubuc Dept of Inf. & Computer Science State Univ. of Rio de Janeiro Rio de Janeiro, Brazil lelia@imc.ufrj.br	Steve Easterbrook Dept of Computer Science University of Toronto Toronto, ON, Canada sme@cs.toronto.ca
Birgit Penzenstadler Institute for Software Research University of California, Irvine Irvine, California, US bpenzens@uci.edu	Norbert Seyff Dept of Informatics University of Zurich Zurich, Switzerland seyff@ifi.uzh.ch	Colin C. Venters School of Computing & Engineering University of Huddersfield Huddersfield, UK c.venters@hud.ac.uk	

Abstract—Sustainability has emerged as a broad concern for society. Many engineering disciplines have been grappling with challenges in how we sustain technical, social and ecological systems. In the software engineering community, for example, maintainability has been a concern for a long time. But too often, these issues are treated in isolation from one another. Misperceptions among practitioners and research communities persist, rooted in a lack of coherent understanding of sustainability, and how it relates to software systems research and practice. This article presents a cross-disciplinary initiative to create a common ground and a point of reference for the global community of research and practice in software and sustainability, to be used for effectively communicating key issues, goals, values and principles of sustainability design for software-intensive systems. The centerpiece of this effort is the *Karlskrona Manifesto for Sustainability Design*, a vehicle for a much needed conversation about sustainability within and beyond the software community, and an articulation of the fundamental principles underpinning design choices that affect sustainability. We describe the motivation for developing this manifesto, including some considerations of the genre of the manifesto as well as the dynamics of its creation. We illustrate the collaborative reflective writing process and present the current edition of the manifesto itself. We assess immediate implications and applications of the articulated principles, compare these to current practice, and suggest future steps.

I. INTRODUCTION

It is clear that society is facing major sustainability challenges that require long-term, joined-up thinking. How do we sustain our technical infrastructures, given how much we rely on them for everything from communication and navigation through to storing health records, identifying security threats, and keeping the lights on? How do we sustain prosperity in society, given the erosion of trust in our political institutions and a growing inequality in ownership of resources? And, above all, how do we sustain the planetary systems that support life on earth, in the face of accumulation of pollutants, species loss, and accelerating climate change?

The discipline of Software Engineering (SE) has a major role to play in sustainability, because of the extent to which software systems mediate so many aspects of our lives. However, software practice has a tendency to focus only on the immediate effects and tangible benefits of software products and platforms. SE research has, for the most part, focused on increasing the reliability, efficiency and cost-benefit relation of software products for their owners, through a focus on processes, methods, models and techniques to create, verify and validate software systems and keep them operational.

The lack of long-term thinking in software engineering research and practice has been critiqued throughout the history of the discipline. For example, software maintenance and evolution were raised as concerns even at the very first software engineering conference [1]. Since then, efforts to increase the maintainability of software products and facilitate their evolution have often focused on improving architecture, decreasing lifecycle costs and managing technical debt [2]. Neumann has criticized the lack of long-term thinking over security considerations in SE [3]. For our digital information assets, some now speak of a ‘digital dark age’ [4], where, having discarded analog media in preference for digital, we now find that many of these assets become unreadable, due, in part, to the rapid lifecycles of software technology.

While progress has been made on design for maintainability of software *per se*, considerations that extend beyond immediate software product qualities and user benefits are generally treated as secondary concerns: optional qualities to be addressed only after the system under design has been shown to be a success in terms of technical and/or marketing criteria. The larger impact of software artefacts on society and the natural environment is not routinely analyzed. But by trading off longer-term sustainability questions for shorter-term success criteria, we accumulate threats to sustainability. We argue that this trade-off itself is unnecessary. As Neumann

(Becker, 2015)

Green Software Engineering

- What is it?

E-waste
Energy consumption

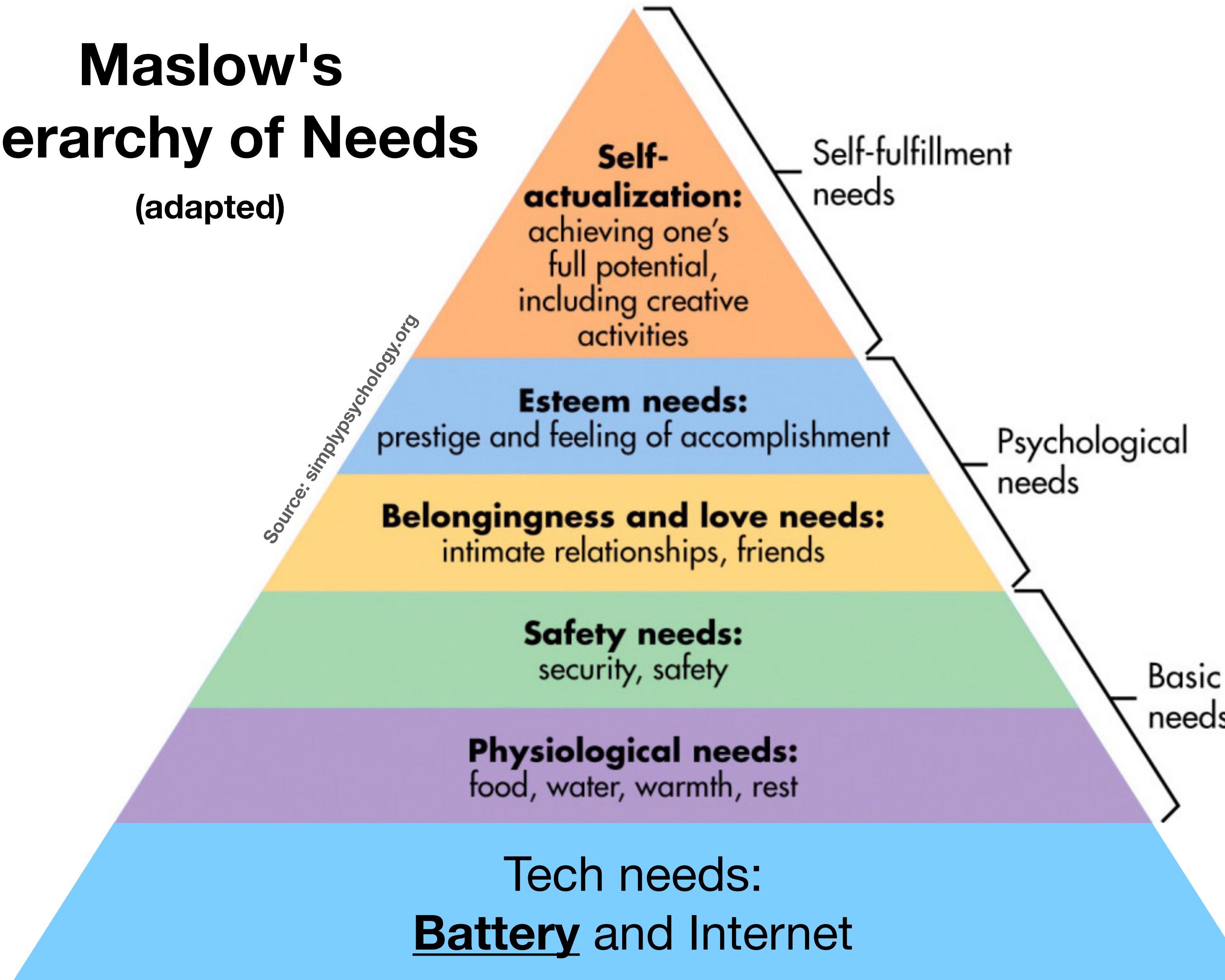
Outline

- Bio ✓
- Software Sustainability ✓
- What is Green Software? ✓
- Motivation for Green Software
- Assignment (teaser)
- How to measure Energy Efficiency
- Energy patterns
- Real Cases
- Programming languages
- Assignment



Maslow's Hierarchy of Needs

(adapted)



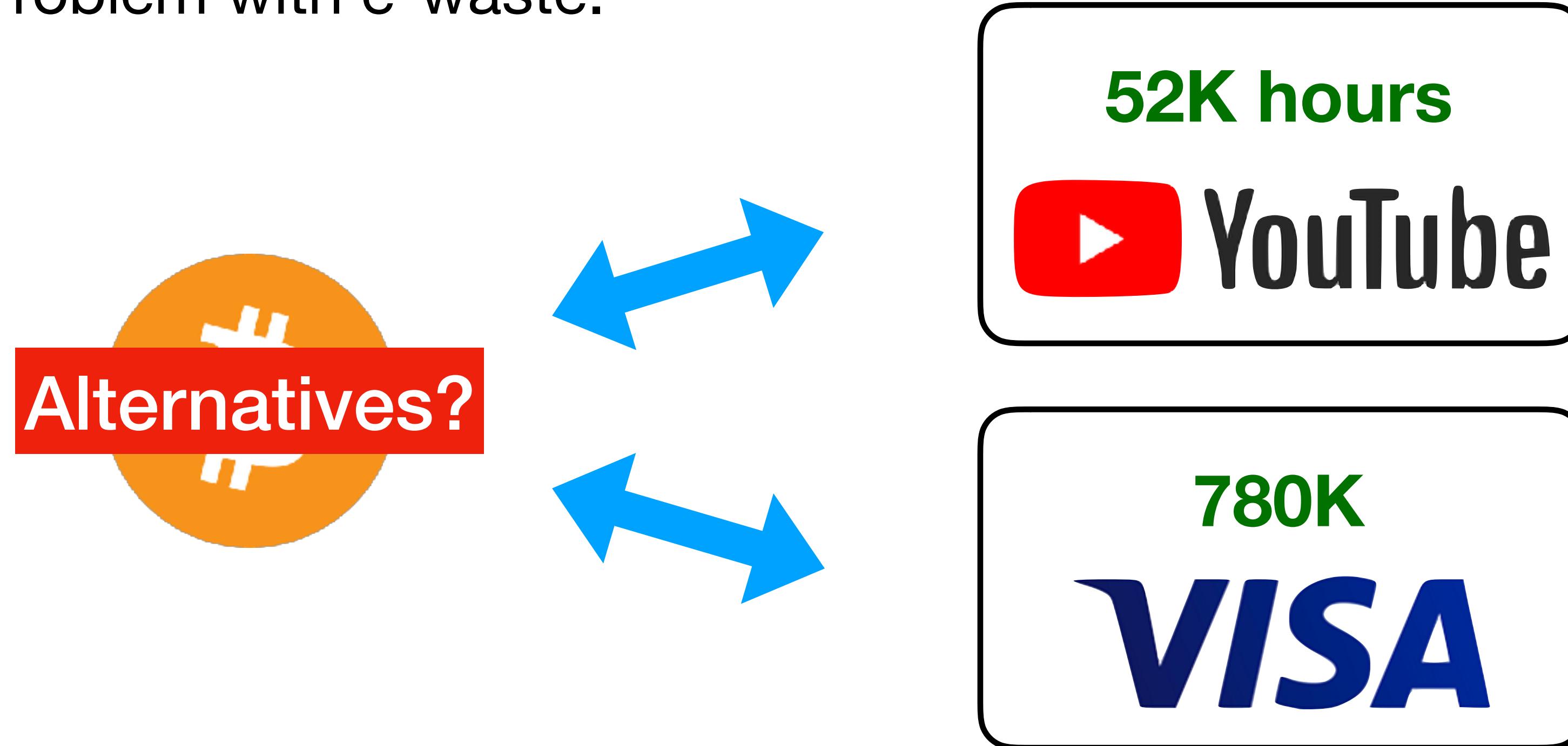
Bitcoin example

- No central authority – consensus algorithm.
- One transaction – multiple agents to compute a hash that validates the transaction.
- Several discussions regarding social sustainability.
- **How environmental impact of bitcoin transactions compare to traditional centralised transactions?**



Bitcoin example

- Annual energy consumption equivalent to Chile's
- Problem with e-waste.



Training Neural Networks

- Deep Learning in NLP.
- Training and tuning an NLP model is comparable to the **CO₂ emission** of a **normal car** throughout its lifetime.
- Researchers should prioritize developing efficient models and hardware.



(Strubell, 2019)

Energy and Policy Considerations for Deep Learning in NLP

Emma Strubell Ananya Ganesh Andrew McCallum
College of Information and Computer Sciences
University of Massachusetts Amherst
{strubell, aganesh, mcallum}@cs.umass.edu

Abstract

Recent progress in hardware and methodology for training neural networks has ushered in a new generation of large networks trained on abundant data. These models have obtained notable gains in accuracy across many NLP tasks. However, these accuracy improvements depend on the availability of exceptionally large computational resources that necessitate similarly substantial energy consumption. As a result these models are costly to train and develop, both financially, due to the cost of hardware and electricity or cloud compute time, and environmentally, due to the carbon footprint required to fuel modern tensor processing hardware. In this paper we bring this issue to the attention of NLP researchers by quantifying the approximate financial and environmental costs of training a variety of recently successful neural network models for NLP. Based on these findings, we propose actionable recommendations to reduce costs and improve equity in NLP research and practice.

Consumption	CO ₂ e (lbs)
Air travel, 1 person, NY↔SF	1984
Human life, avg, 1 year	11,023
American life, avg, 1 year	36,156
Car, avg incl. fuel, 1 lifetime	126,000
Training one model (GPU)	
NLP pipeline (parsing, SRL)	39
w/ tuning & experiments	78,468
Transformer (big)	192
w/ neural arch. search	626,155

Table 1: Estimated CO₂ emissions from training common NLP models, compared to familiar consumption.¹

NLP models could be trained and developed on a commodity laptop or server, many now require multiple instances of specialized hardware such as GPUs or TPUs, therefore limiting access to these highly accurate models on the basis of finances.

Even when these expensive computational resources are available, model training also incurs a substantial cost to the environment due to the energy required to power this hardware for weeks or months at a time. Though some of this energy may come from renewable or carbon credit-offset resources, the high energy demands of these models are still a concern since (1) energy is not currently derived from carbon-neutral sources in many locations, and (2) when renewable energy is available, it is still limited to the equipment we have to produce and store it, and energy spent training a neural network might better be allocated to heating a family's home. It is estimated that we must cut carbon emissions by half over the next decade to deter escalating rates of natural disaster, and based on the estimated CO₂ emissions listed in Table 1,

¹Sources: (1) Air travel and per capita consumption: <https://bit.ly/2Bw0xNe>; (2) car lifetime: <https://bit.ly/2Qhr0wL>.

Software Architectural Decisions

- 28,895 -> 144 relevant papers
- Domain-specific architectural decisions (such as mobile).
- Quality attributes (such as reliability or scalability). **Security, reliability, usability, scalability, evolvability, safety.**
- Uncertainty in decision-making, and group architectural decisions.

Information and Software Technology 56 (2014) 850–859

Contents lists available at ScienceDirect
 Information and Software Technology
journal homepage: www.elsevier.com/locate/insof

Past and future of software architectural decisions – A systematic mapping study

Dan Tofan ^{a,*, b}, Matthias Galster ^b, Paris Avgeriou ^a, Wes Schuitama ^a

^a Department of Mathematics and Computing Science, University of Groningen, Nijenborgh 9, 9742 AG Groningen, Netherlands
^b Department of Computer Science and Software Engineering, University of Canterbury, Private Bag 4900, Christchurch 8140, New Zealand

ARTICLE INFO

Article history:
Received 5 August 2013
Received in revised form 14 March 2014
Accepted 26 March 2014
Available online 31 March 2014

Keywords:
Software architecture
Architectural decisions
Systematic mapping study

ABSTRACT

Context: The software architecture of a system is the result of a set of architectural decisions. The topic of architectural decisions in software engineering has received significant attention in recent years. However, no systematic overview exists on the state of research on architectural decisions.

Objective: The goal of this study is to provide a systematic overview of the state of research on architectural decisions. Such an overview helps researchers reflect on previous research and plan future research. Furthermore, such an overview helps practitioners understand the state of research, and how research results can help practitioners in their architectural decision-making.

Method: We conducted a systematic mapping study, covering studies published between January 2002 and January 2012. We defined six research questions. We queried six reference databases and obtained an initial result set of 28,895 papers. We followed a search and filtering process that resulted in 144 relevant papers.

Results: After classifying the 144 relevant papers for each research question, we found that current research focuses on documenting architectural decisions. We found that only several studies describe architectural decisions from the industry. We identified potential future research topics: domain-specific architectural decisions (such as mobile), achieving specific quality attributes (such as reliability or scalability), uncertainty in decision-making, and group architectural decisions. Regarding empirical evaluations of the papers, around half of the papers use systematic empirical evaluation approaches (such as surveys, or case studies). Still, few papers on architectural decisions use experiments.

Conclusion: Our study confirms the increasing interest in the topic of architectural decisions. This study helps the community reflect on the past ten years of research on architectural decisions. Researchers are informed a number of promising future research directions, while practitioners learn what existing papers offer.

* Corresponding author. Tel.: +31 503 633 968.
E-mail address: d.tofan@rug.nl (D. Tofan).

<http://dx.doi.org/10.1016/j.insof.2014.03.008>
0890-5909/\$ 2014 Elsevier B.V. All rights reserved.

Contents

1. Introduction	851
2. Research methodology	852
2.1. Research directives	852
2.1.1. Protocol definition	852
2.1.2. Generic decision literature survey	853
2.1.3. Research questions definition	853
2.1.3.1. Research questions derived from software architecture literature	853
2.1.3.2. Research questions derived from generic decision literature	854
3. Data collection	854
3.1. Source selection and search string	854
3.2. Inclusion and exclusion criteria	855
3.3. Search process	856

(Tofan, 2014)

Green Field

- There is no awareness of the energy consumption.
- There is little information about the energy consumption of our decisions and practices as software architects and developers.
- Little is known about our footprint as users and developers.
E.g, watch a movie in streaming platforms.

Greenpeace Report

- IT sector consumes 7% of global electricity (2017).
- “*The continued **lack of transparency** by many companies regarding their **energy demand** and the **supply of electricity** powering their data centers remains a significant threat to the sector’s long-term **sustainability**.*”
- Report provides an analysis of environmental sustainability of tech providers in different angles. **Transparency, Commitment, Energy Efficiency, Renewable Procurement, Advocacy.**



Gary Cook, Jude Lee, Tamina Tsai, Ada Kongn, John Deans, Brian Johnson, Elizabeth Jardim, and Brian Johnson.
2017. Clicking Clean: Who is winning the race to build a green internet? Technical report, Greenpeace.

	Final Grade	 Clean Energy Index	 Natural Gas	 Coal	 Nuclear
 Adobe	B	23%	37%	23%	11%
 Alibaba.com™	D	24%	3%	67%	3%
 amazon.com web services	C	17%	24%	30%	26%
 Apple	A	83%	4%	5%	5%
 Baidu 百度	F	24%	3%	67%	3%
 f	A	67%	7%	15%	9%
 Google	A	56%	14%	15%	10%
 hp	C	50%	17%	27%	5%

	Final Grade	 Clean Energy Index	 Natural Gas	 Coal	 Nuclear
 IBM	C	29%	29%	27%	15%
 Microsoft	B	32%	23%	31%	10%
 NAVER	C	2%	19%	39%	31%
 ORACLE®	D	8%	26%	36%	25%
 salesforce	B	43%	12%	16%	15%
 삼성SDS	D	11%	19%	29%	31%
 Tencent 腾讯	F	24%	3%	67%	3%

(Greenpeace, 2017)

(Side note)

Video Streaming

	Final Grade	Clean Energy Index	Natural Gas	Coal	Nuclear
Afreeca.com	F	2%	19%	39%	31%
Amazon Prime	C	17%	24%	30%	26%
HBO	D	22%	20%	25%	25%
Hulu	F	20%	30%	29%	20%
Netflix	D	17%	24%	30%	26%
Poqq.co.kr	F	2%	19%	39%	31%
Vevo	F	27%	15%	32%	26%
Vimeo	D	47%	13%	20%	19%
YouTube	A	56%	15%	14%	10%

Music/Audio Streaming

	Final Grade	Clean Energy Index	Natural Gas	Coal	Nuclear
iTunes	A	83%	4%	5%	5%
NPR	F	17%	24%	30%	26%
Pandora	F	13%	32%	20%	27%
SoundCloud	F	17%	24%	30%	26%
Spotify	D	56%	15%	14%	10%
Podbbang	F	2%	19%	39%	31%

Sources of Energy Consumption

- Execution
- Development
- Infrastructure

Checkpoint

- Green software engineering is a green field.
- Some SOA technologies are unsustainable
- Bring energy efficiency to the equation (**awareness**)
- Energy efficiency can be tackled at different levels (execution, development, infrastructure)
- Transparency of providers
- The decisions of Software Engineers make a difference on environmental sustainability

Assignment

- Look into the change history of the project and **find code changes** that are related to green computing. Present and discuss the rationale behind those changes.
- Suggest **energy improvements** to be implemented in the project (development, source, infrastructure). Implement them, if possible.
- **Measure** the energy consumption of potential **hotspots**.
- **Output:** 1–2 page report with all the rationale behind the study
- **Critical thinking.**



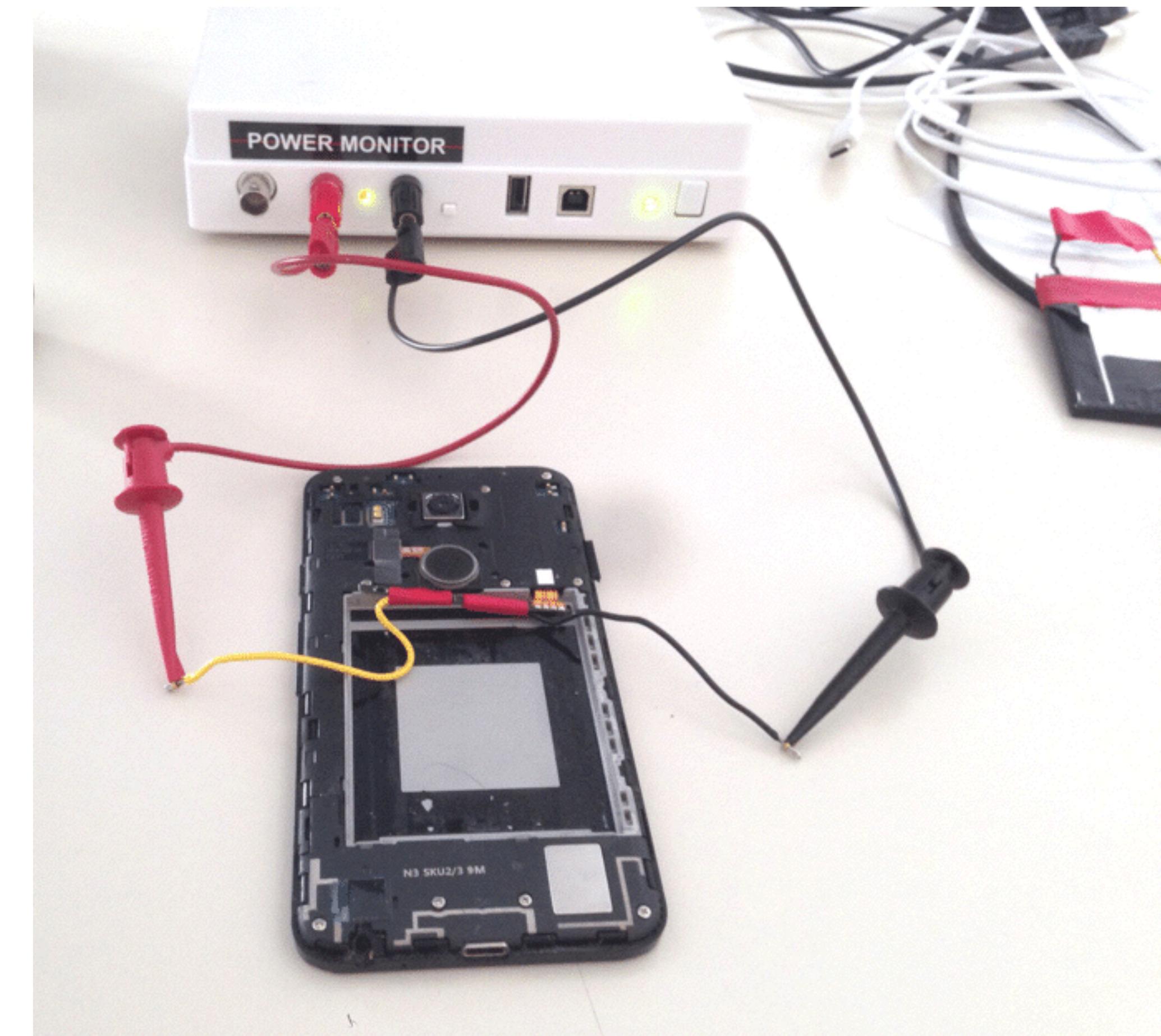
Technical
⚠

How to Measure Energy Consumption

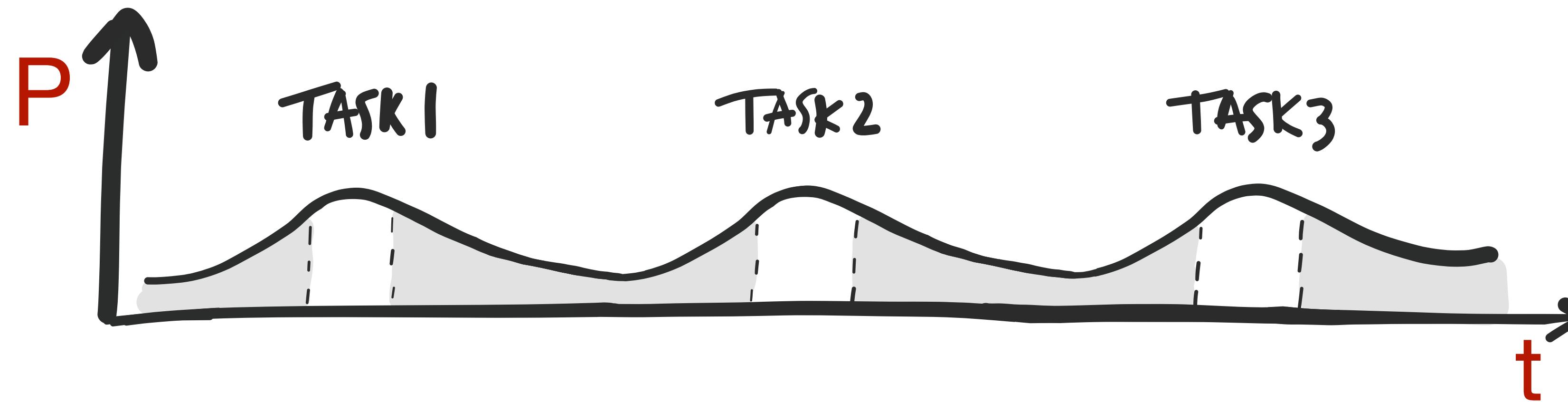
1. Create a scenario.
2. Execute and **collect power data**.
3. Implement energy improvement.
4. Execute and collect power data.
5. Analyse and compare results.

Collect Power Data

- Electricity bill 😷
- Execution time
- Estimation tools (Intel® RAPL)
<https://01.org/rapl-power-meter>
- Power Monitors (Monsoon)



Tail Energy Consumption



$$W = P \cdot \Delta t$$

$$W = \int_{t_1}^{t_2} P(t) \cdot dt$$

- Measuring energy consumption is difficult!
- Solutions?

Energy Patterns for Mobile

A screenshot of a web browser displaying the 'Energy Patterns for Mobile Apps' website. The page title is 'Energy Patterns for Mobile Apps'. Below it, a green banner contains a leaf icon and the text: 'A visualization with prevalence and co-occurrence of patterns can be found [here](#). News: This catalog has been accepted to the *Journal of Empirical Software Engineering*. Check out the [preprint](#).'. A link '← show all patterns' is visible. The main content section is titled 'Dark UI Colors' with the sub-section 'Context'. It includes a diagram showing two smartphones, one with a light background and one with a dark background, both displaying a news feed. The text explains that screens are a major source of power consumption on mobile devices, particularly for AMOLED screens. The 'Solution' section provides a UI with dark background colors, noting its benefit for AMOLED screens and the option for users to choose between light and dark themes. A 'Display a menu' button is at the bottom left.



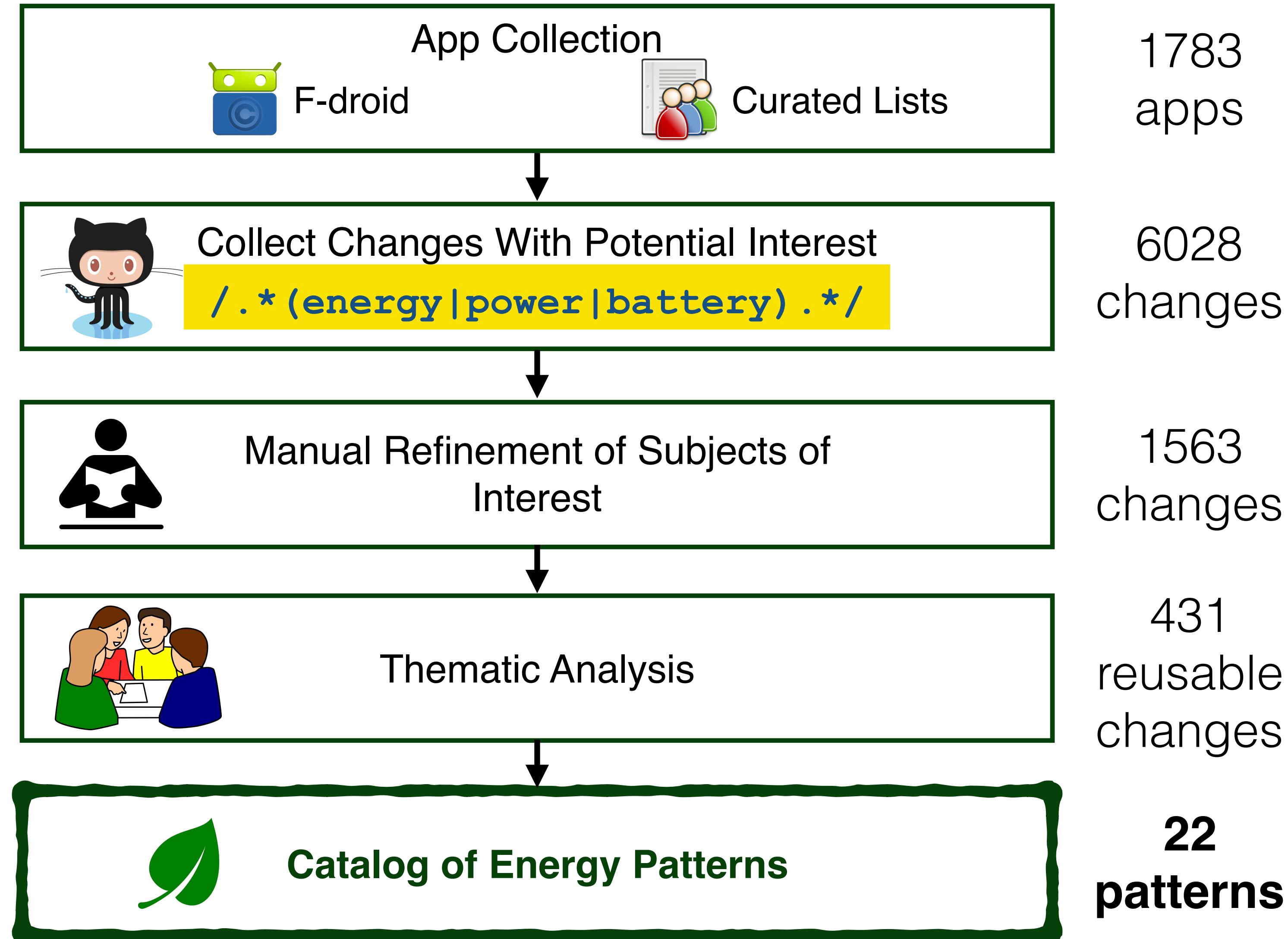
<https://tqrg.github.io/energy-patterns/>

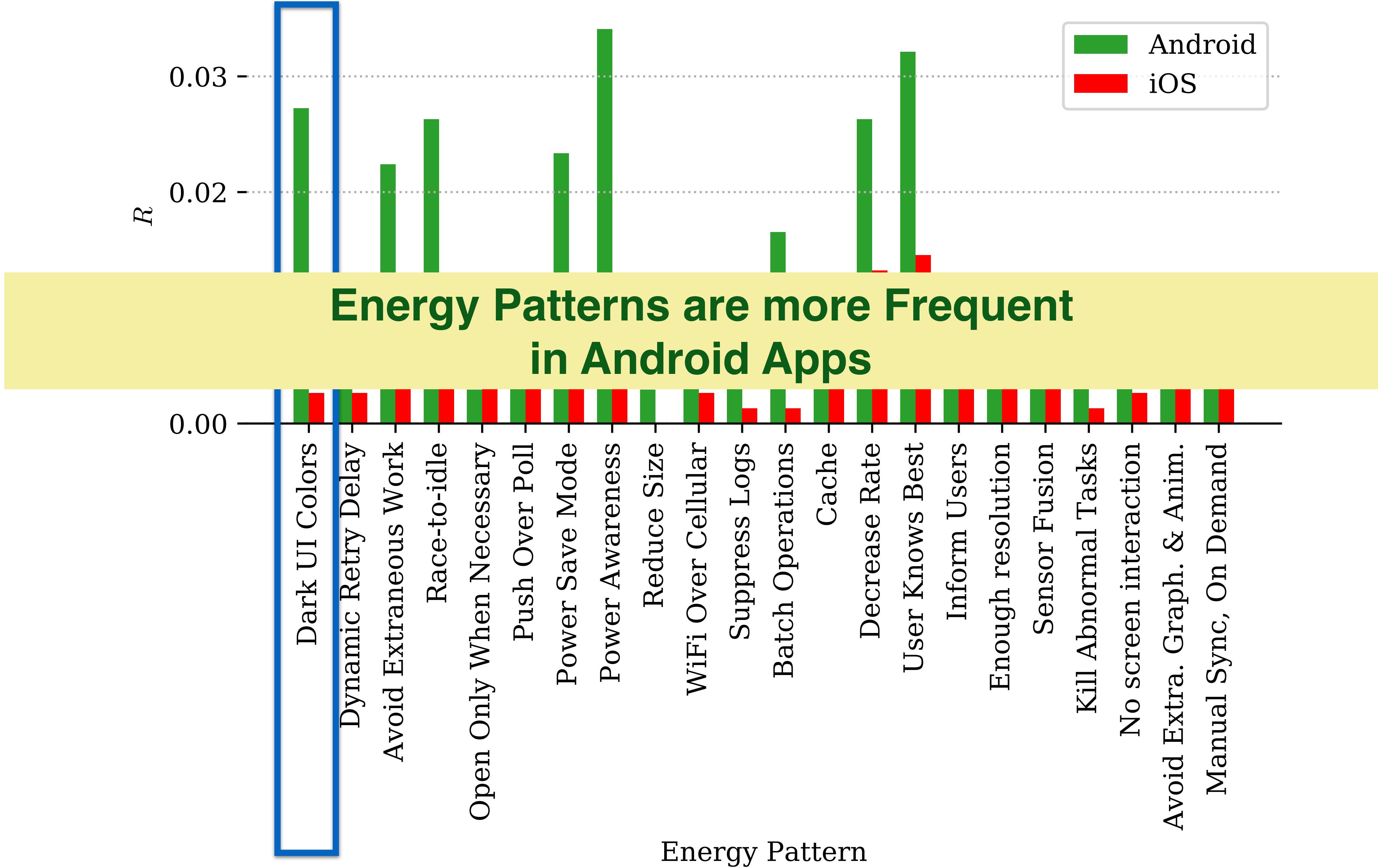
Energy Patterns

- Dynamic Retry Delay
- Avoid Extraneous Graphics and Animations
- Push over Poll
- Inform Users

Cover them in the website

Methodology



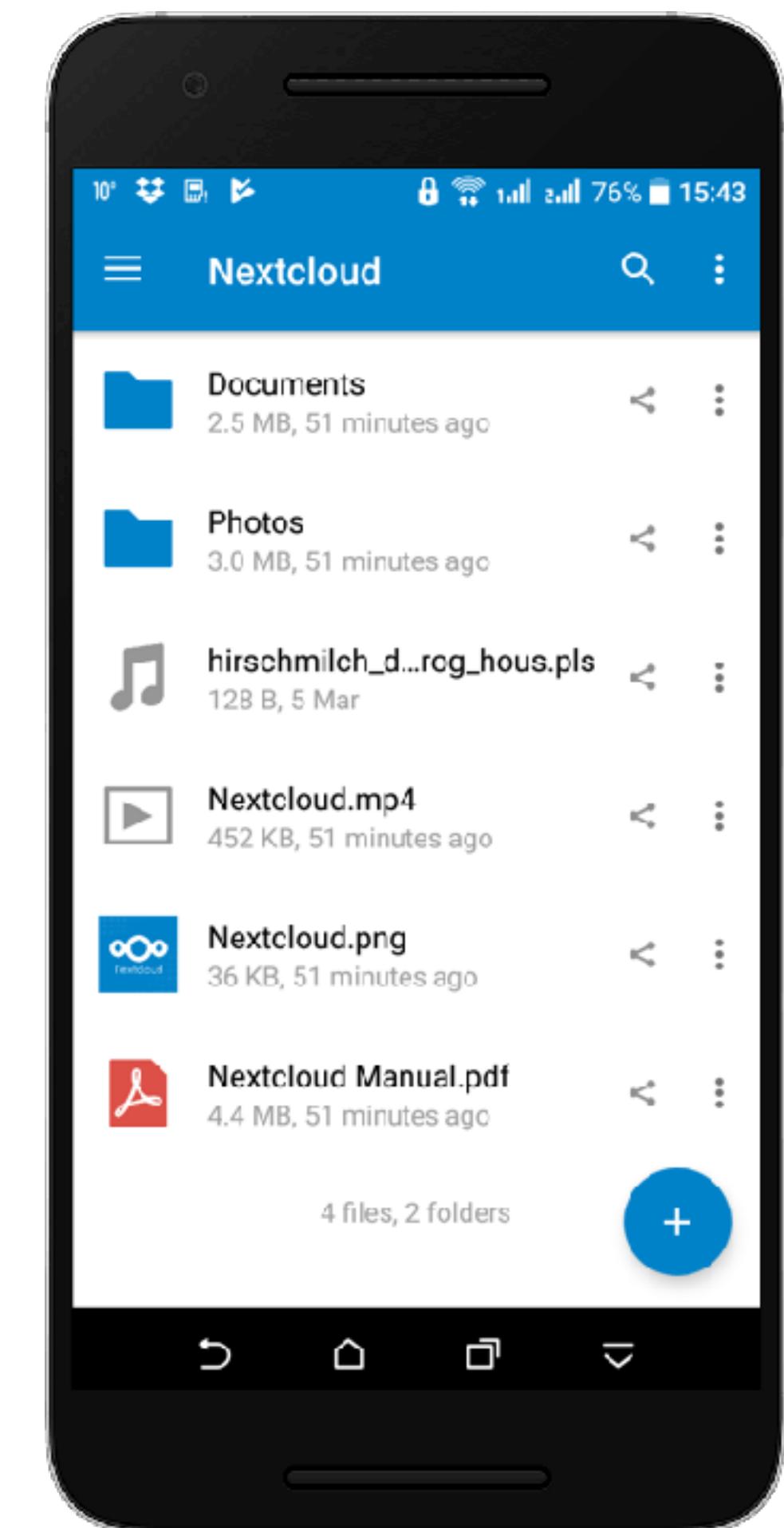
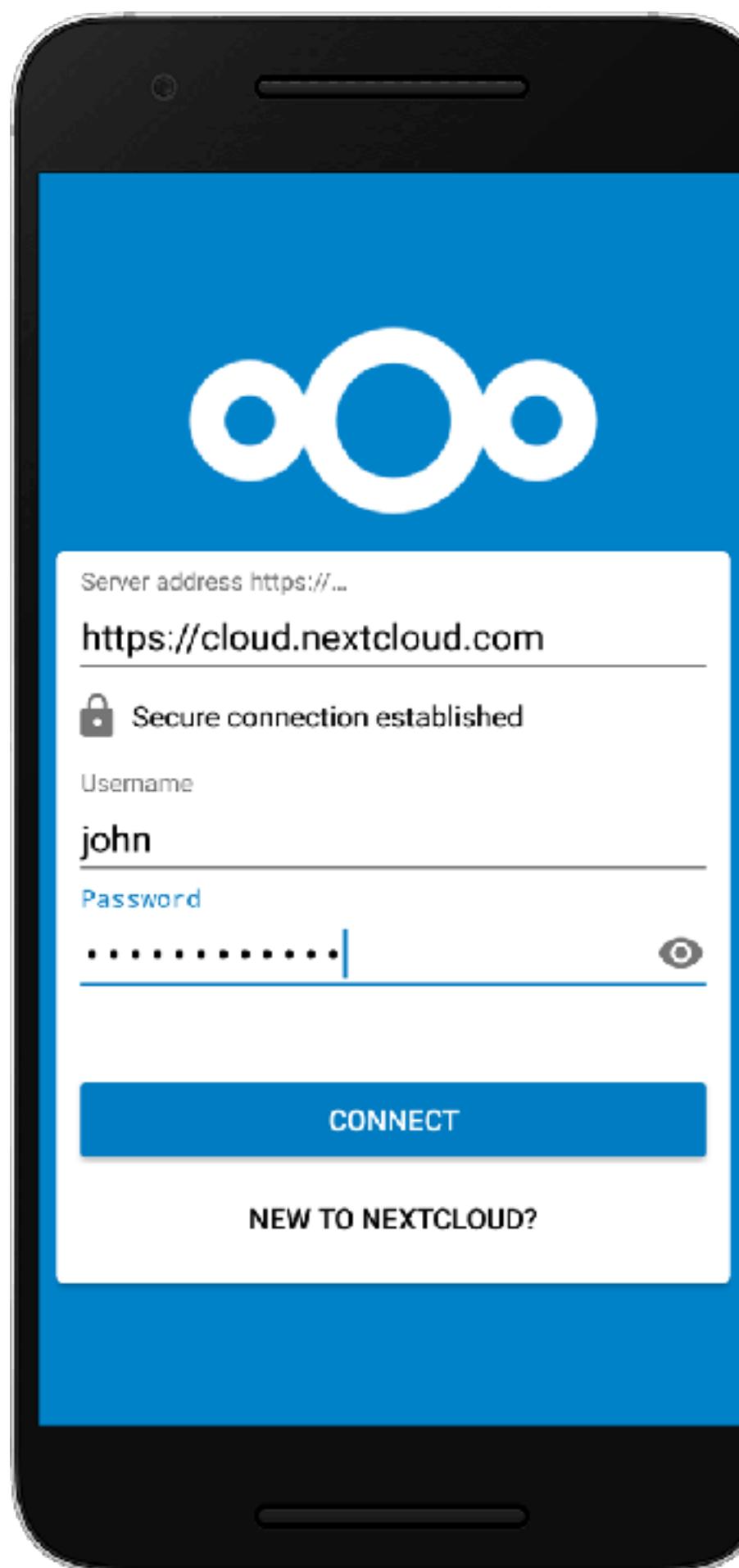


Example case: Nextcloud



Nextcloud

FOSS →



Example case: Nextcloud

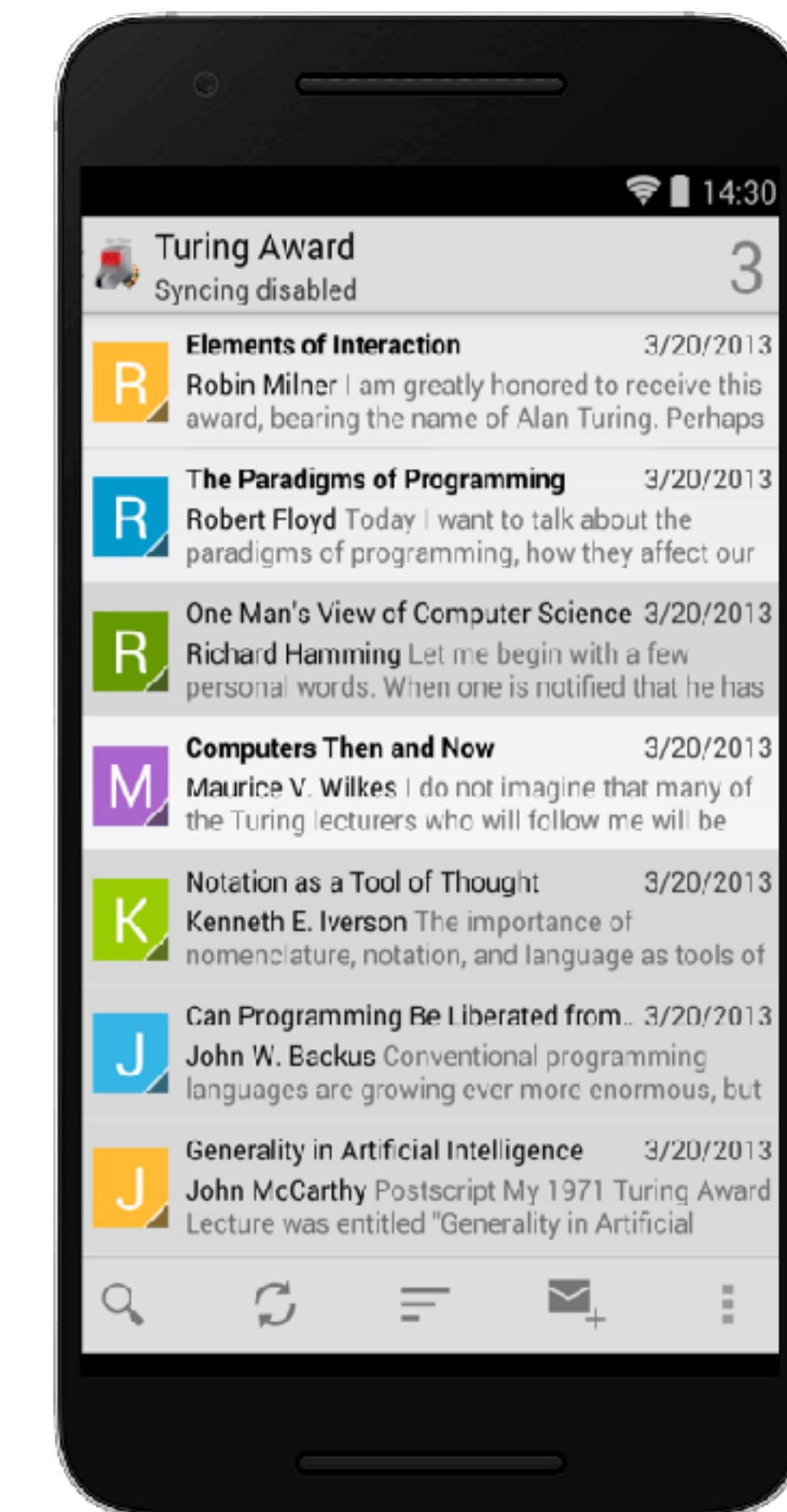
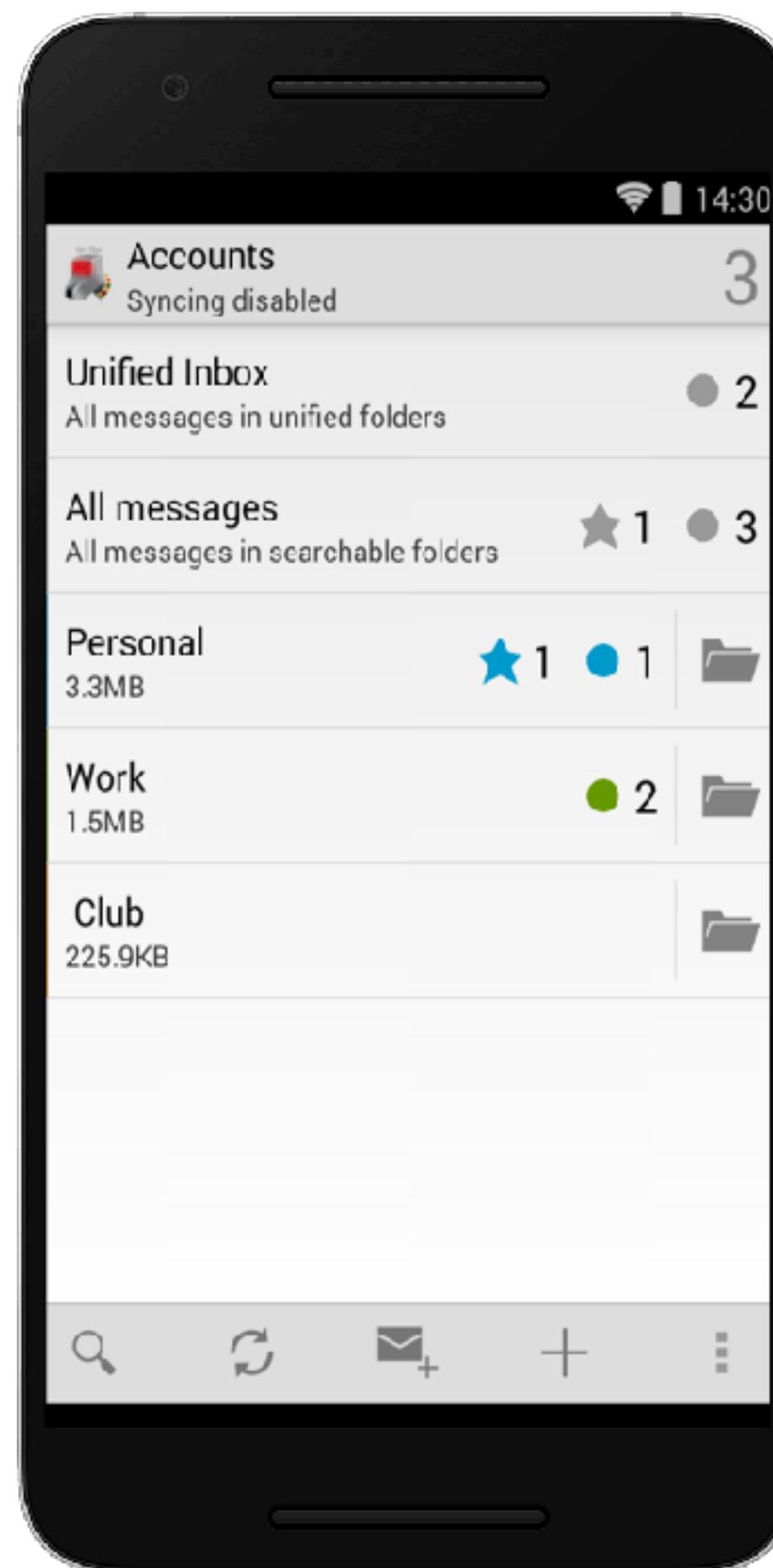
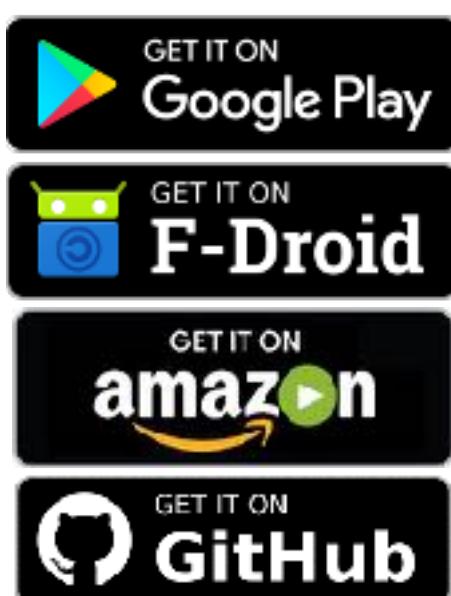
- Users complain that sometimes they go on a trip and Nextcloud drains their battery. Users consider uninstalling the app when battery life is essential.

Solutions?

Example case: Nextcloud

- Trade-offs?

Example case: K-9 mail



Example case: K-9 mail

- Some users noticed that K-9 mail was spending more energy than usual.



Example case: Nextcloud

- Trade-offs?

Which programming languages are most energy efficient?

<https://sites.google.com/view/energy-efficiency-languages>

35

Energy Efficiency across Programming Languages

How Do Energy, Time, and Memory Relate?

Rui Pereira
HASLab/INESC TEC
Universidade do Minho, Portugal
rui.pereira@di.uminho.pt

Jácome Cunha
NOVA LINCS, DI, FCT
Univ. Nova de Lisboa, Portugal
jacome@fct.unl.pt

Marco Couto
HASLab/INESC TEC
Universidade do Minho, Portugal
marco.l.couto@inesctec.pt

João Paulo Fernandes
Release/LISP, CISUC
Universidade de Coimbra, Portugal
jpf@dei.uc.pt

Francisco Ribeiro, Rui Rua
HASLab/INESC TEC
Universidade do Minho, Portugal
fribeiro@di.uminho.pt
rrua@di.uminho.pt

João Saraiva
HASLab/INESC TEC
Universidade do Minho, Portugal
saraiva@di.uminho.pt

Abstract

This paper presents a study of the runtime, memory usage and energy consumption of twenty seven well-known software languages. We monitor the performance of such languages using ten different programming problems, expressed in each of the languages. Our results show interesting findings, such as, slower/faster languages consuming less/more energy, and how memory usage influences energy consumption. We show how to use our results to provide software engineers support to decide which language to use when energy efficiency is a concern.

CCS Concepts → Software and its engineering → Software performance; General programming languages;

Keywords Energy Efficiency, Programming Languages, Language Benchmarking, Green Software

ACM Reference Format:

Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate?. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17)*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3136014.3136031>

1 Introduction

Software language engineering provides powerful techniques and tools to design, implement and evolve software languages. Such techniques aim at improving programmers

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SLE'17, October 23–24, 2017, Vancouver, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5525-0/17/10...\$15.00
<https://doi.org/10.1145/3136014.3136031>

productivity - by incorporating advanced features in the language design, like for instance powerful modular and type systems - and at efficiently execute such software - by developing, for example, aggressive compiler optimizations. Indeed, most techniques were developed with the main goal of helping software developers in producing faster programs. In fact, in the last century *performance* in software languages was in almost all cases synonymous of *fast execution time* (embedded systems were probably the single exception).

In this century, this reality is quickly changing and software energy consumption is becoming a key concern for computer manufacturers, software language engineers, programmers, and even regular computer users. Nowadays, it is usual to see mobile phone users (which are powerful computers) avoiding using CPU intensive applications just to save battery/energy. While the concern on the computers' energy efficiency started by the hardware manufacturers, it quickly became a concern for software developers too [28]. In fact, this is a recent and intensive area of research where several techniques to analyze and optimize the energy consumption of software systems are being developed. Such techniques already provide knowledge on the energy efficiency of data structures [15, 27] and android language [25], the energy impact of different programming practices both in mobile [18, 22, 31] and desktop applications [26, 32], the energy efficiency of applications within the same scope [2, 17], or even on how to predict energy consumption in several software systems [4, 14], among several other works.

An interesting question that frequently arises in the software energy efficiency area is whether a *faster program is also an energy efficient program*, or not. If the answer is yes, then optimizing a program for speed also means optimizing it for energy, and this is exactly what the compiler construction community has been hardly doing since the very beginning of software languages. However, energy consumption does not depends only on execution time, as shown in the equation $E_{nrgy} = T_{ime} \times P_{ower}$. In fact, there are several research works showing different results regarding

256

(Pereira, 2017)

The Computer Language Benchmarks Game

<https://benchmarksgame-team.pages.debian.net/benchmarksgame/which-programs-are-fastest.html>

Benchmark	Description	Input
n-body	Double precision N-body simulation	50M
fannkuch-redux	Indexed access to tiny integer sequence	12
spectral-norm	Eigenvalue using the power method	5,500
mandelbrot	Generate Mandelbrot set portable bitmap file	16,000
pidigits	Streaming arbitrary precision arithmetic	10,000
regex-redux	Match DNA 8mers and substitute magic patterns	fasta output
fasta	Generate and write random DNA sequences	25M
k-nucleotide	Hashtable update and k-nucleotide strings	fasta output
reverse-complement	Read DNA sequences, write their reverse-complement	fasta output
binary-trees	Allocate, traverse and deallocate many binary trees	21
chameneos-redux	Symmetrical thread rendezvous requests	6M
meteor-contest	Search for solutions to shape packing puzzle	2,098
thread-ring	Switch from thread to thread passing one token	50M

(Pereira, 2017)

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

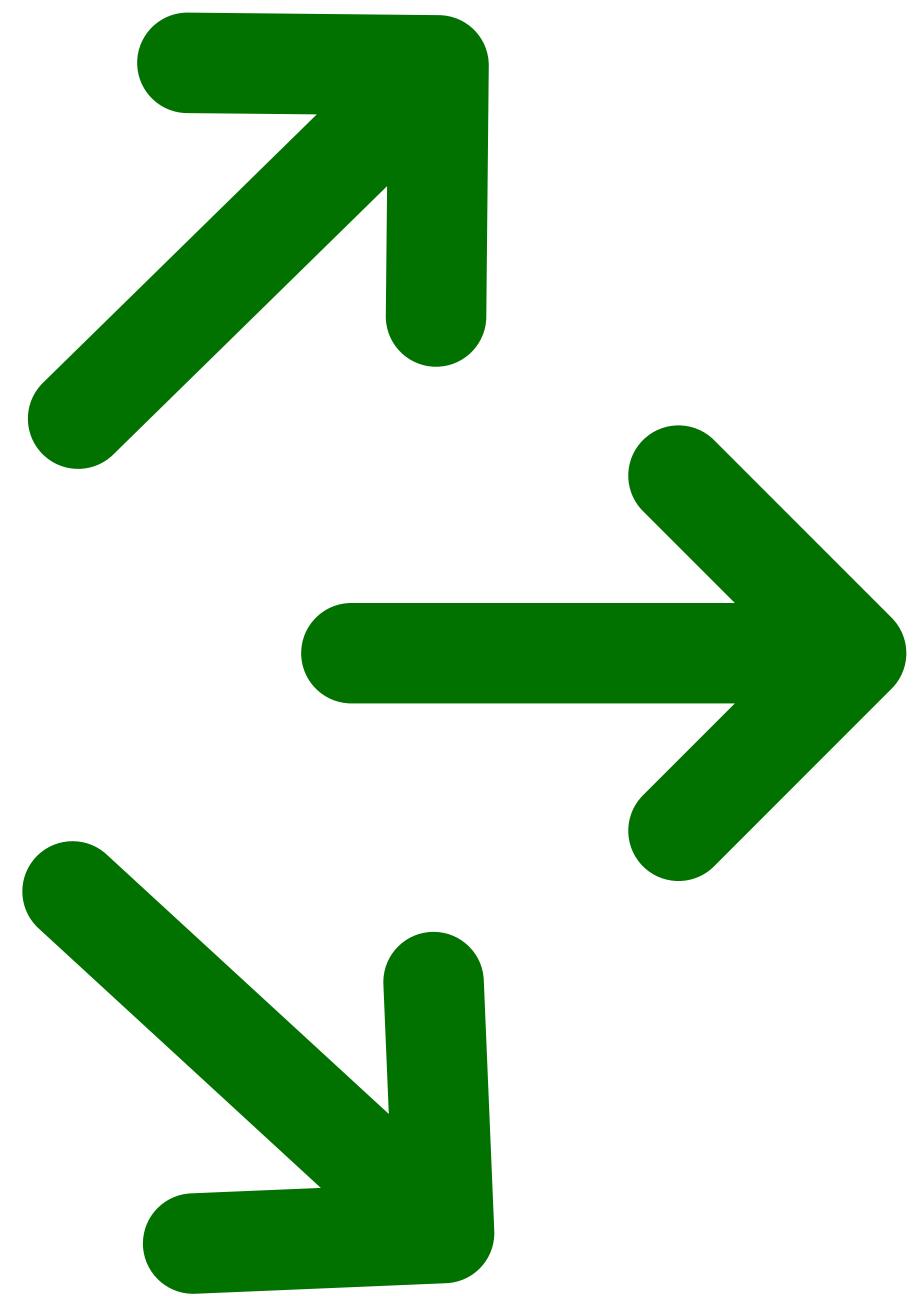
(Pereira, 2017)

Time & Memory	Energy & Time	Energy & Memory	Energy & Time & Memory
C • Pascal • Go	C	C • Pascal	C • Pascal • Go
Rust • C++ • Fortran	Rust	Rust • C++ • Fortran • Go	Rust • C++ • Fortran
Ada	C++	Ada	Ada
Java • Chapel • Lisp • Ocaml	Ada	Java • Chapel • Lisp	Java • Chapel • Lisp • Ocaml
Haskell • C#	Java	OCaml • Swift • Haskell	Swift • Haskell • C#
Swift • PHP	Pascal • Chapel	C# • PHP	Dart • F# • Racket • Hack • PHP
F# • Racket • Hack • Python	Lisp • Ocaml • Go	Dart • F# • Racket • Hack • Python	JavaScript • Ruby • Python
JavaScript • Ruby	Fortran • Haskell • C#	JavaScript • Ruby	TypeScript • Erlang
Dart • TypeScript • Erlang	Swift	TypeScript	Lua • JRuby • Perl
JRuby • Perl	Dart • F#	Erlang • Lua • Perl	
Lua	JavaScript	JRuby	
	Racket		
	TypeScript • Hack		
	PHP		
	Erlang		
	Lua • JRuby		
	Ruby		

(Pereira, 2017)

Green Open Field

- Integrate systems with **energy consumption feedback**. **Inform users**
- Energy-efficiency leverage at different levels. **API**, **programming language**, **IDE**, **user**, **developer**, etc.
- Impact of different architectures. **Controlling for implementation, hardware and feature set is not trivial.**



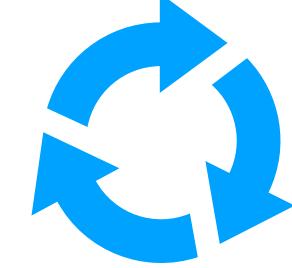
Wrap-up

- What is Sustainable Software
- What is Green Software?
- How can we measure energy consumption?
- What are the sources of energy consumption in software engineering?
- What is an energy pattern?
- What are the common trade-offs when improving energy efficiency?

Other Research Topics

- Machine Learning Deployment at scale
- ML coding practices
- ML maintainability, technical debt
- ML lifecycle management
- ML documentation generation
- MH in SE

ML, AI, DS, Big Data



Assignment

- Look into the change history of the project and **find code changes** that are related to green computing. Present and discuss the rationale behind those changes.
- Suggest **energy improvements** to be implemented in the project (development, source, infrastructure). Implement them, if possible.
- **Measure** the energy consumption of potential **hotspots**. (use RAPL)

Output: 1–2 page report with all the rationale behind the study

- Critical thinking is a requirement:
 - No assumptions or preconceptions, think outside the script.
 - Is it always possible to reduce energy consumption?
 - What are the trade-offs of improving energy efficiency?
 - What are the implications on UX or business metrics?
 - Would automation tools help?
 - What is missing in the project to leverage energy efficiency?



Technical
⚠