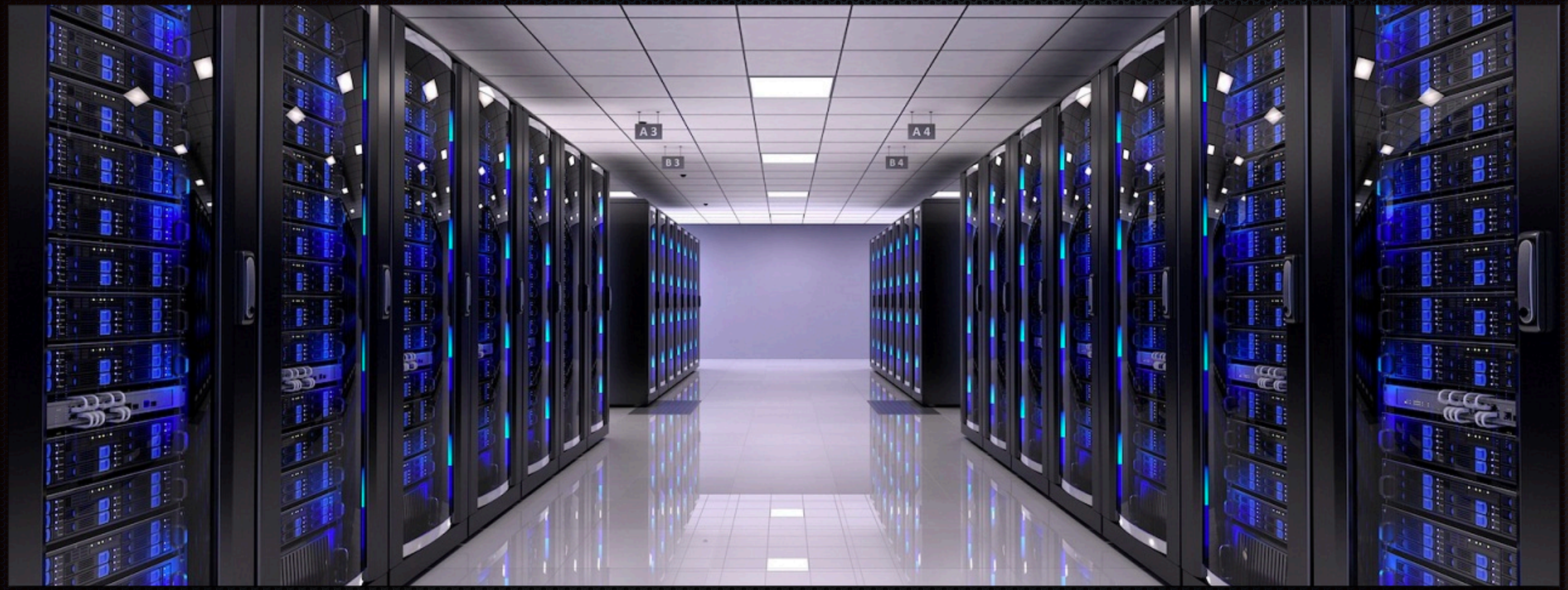


Maîtrise de l’impact environnemental du numérique et des systèmes d’informations

Focus sur les problématiques d’éco-conception logicielle

Séminaire ISEP/ICP du 21 octobre 2021

Renaud Pawlak
CINCHEO, ID Capture - coordinateur de la R&D EASYTEAM (groupe Constellation) sur le bas carbone

Renaud Pawlak

- Ex-chercheur INRIA, ex-responsable du département informatique de l'ISEP, ex-directeur du lab Mantu (IA)
- Co-fondateur de ID Capture : 10 ans de développement de services numériques pour l'industrie de la construction
- Créateur de CINCHEO, R&D développement logiciel responsable (JSweet, SweetHome3D, INI, dLite…)
- Conseiller & coordinateur R&D pour EASYTEAM (groupe Constellation) sur la maîtrise de l'empreinte environnementale du numérique
- Membre de l'initiative Ecolog (IMT Atlantique en partenariat avec l'INR)

Plan

- Pourquoi s'intéresser à l'empreinte environnementale du numérique ?
- Introduction à l'éco-conception logicielle
- L'efficacité énergétique des logiciels
- La maîtrise de la complexité (compilation, abstraction, dépendances)
- L'importance de l'architecture
- Conclusions et conseils pour construire des logiciels plus responsables et plus durables

Pourquoi s’intéresser à l’empreinte environnementale du numérique ?

Motivations

Impacts du numérique

- Aujourd'hui: le numérique représente 4% des émissions mondiales de GES (équivalent au secteur de l'aviation civile)
- Projections 2025: 8% (= voiture individuelle)
  - Progression exponentielle (Big Data, BlockChain, Machine Learning, …)
- Effets indirects de second ordre et multi-sectoriels (par exemple : changement de comportements lié au télé-travail)

Législations en cours autour de la "taxe carbone"
Accords de Paris => -50% d'ici 2030

Le cas français

- En France le numérique représente 2% des impacts eq CO2 (électricité nucléaire)
- Estimé à 7% en 2040

Proposition de loi : la loi REEN

- Formation
- Création de référentiels (impacts et bonnes pratiques) - ADEME *Dinum, le Ministère de la Transition Écologique (MTE), l'Ademe et l'Institut du Numérique Responsable (INR)*
- Lutte contre l'obsolescence programmée (droit à la réparation, droit à l'installation d'autres logiciels)
- Promotion des équipements moins énergivores (datacenters, réseaux)
- Favoriser les usages numériques vertueux
- Stratégie d'aménagement du territoire

Notion d'impact : effets multiples

- Effets de premier ordre (effets directs, cycle de vie, effets de substitution)
  - *Exemple: remplacement des appareils photos et des montres par des iPhones*
- Effets de second ordre : effets "conséquentiels" (effets de passage à l'échelle (loi de Theodore Wright - 1936), effets rebond (paradoxe de Jevons), effets de compensation, …
- Les impacts de second ordre sont rarement cantonnés à un secteur
  - *Exemple : un service de e-commerce aura des effets sur les secteurs du recyclage, de l'emballage, du transport, et de la construction (stockage)*
  - *Exemple : le télé-travail au des impacts sur les transports et sur la construction (et donc sur l'énergie pour le chauffage, c.f. la crise sanitaire récente)*

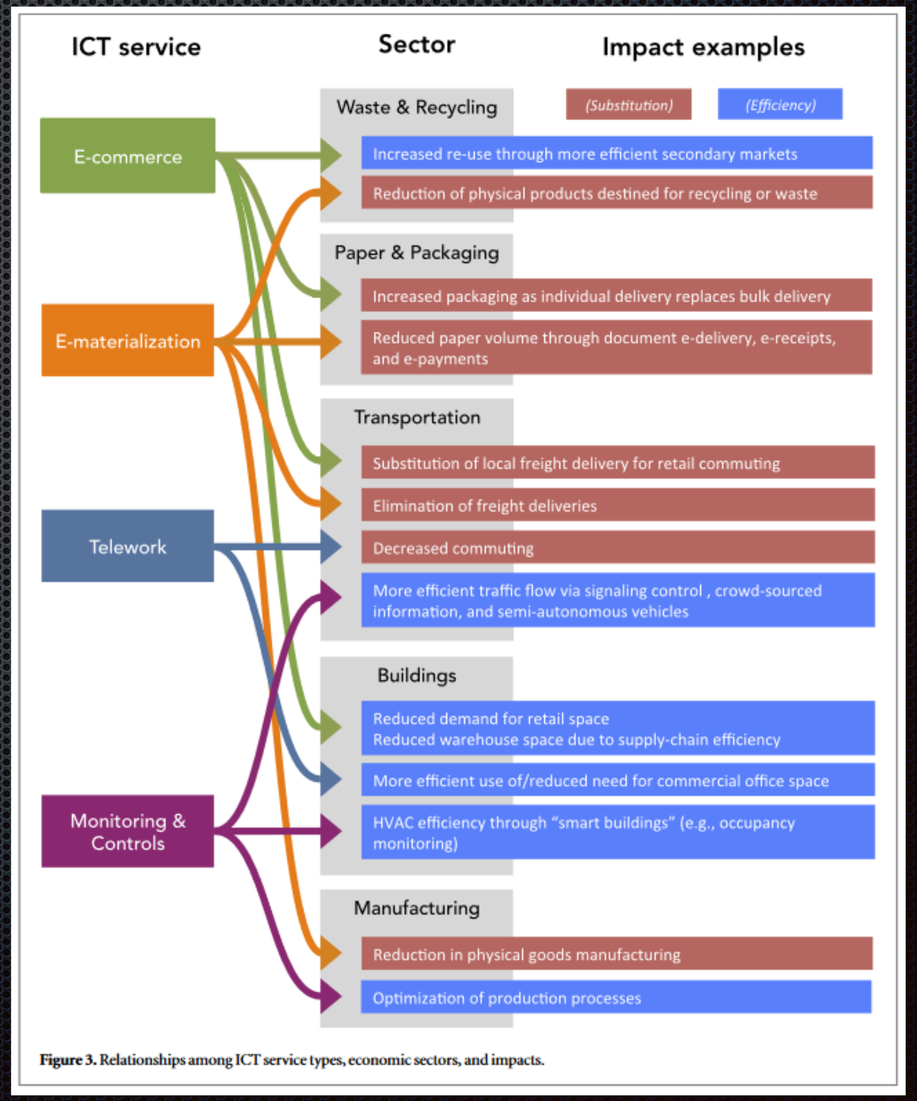


Figure 3. Relationships among ICT service types, economic sectors, and impacts.

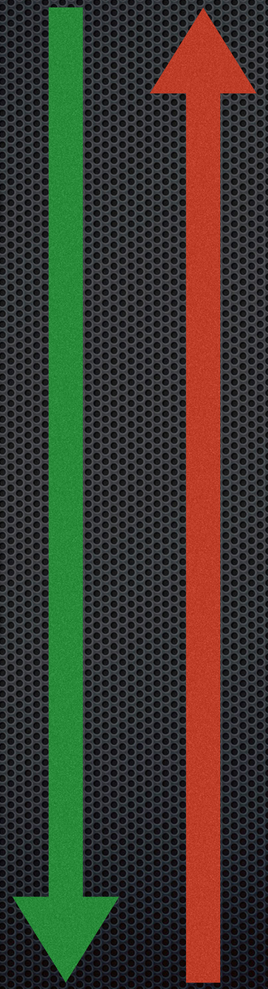
Les effets rebond

- So far, enthusiasm for policies to promote energy efficiency has not been diminished by the Jevons paradox, named after William Stanley Jevons, an economist who made the link between increased technological efficiency (in his case, regarding the use of coal) and consumption, or by the conclusion: **that the effect of improving the efficiency of a factor of production, like energy, is to lower its implicit price and hence make its use more affordable, thus leading to greater use. [1]**
- Exemples d'effets de second ordre (effets rebonds, paradoxe de Jevons)
  - *Exemple: taille des écrans LCD*
  - *Exemple: en Angleterre ou en Allemagne, l'isolation des maisons n'a pas diminué la consommation [2]*

[1] Herring, H. (2006). Energy efficiency – a critical view. *Energy*, *31*, 10–20. doi: 10.1016/j.energy.2004.04.055

[2] Hamilton, I. G., Summerfield, A. J., & Shipworth, D., et al. (2016). Energy efficiency uptake and energy savings in English houses: A cohort study. *Energy and Buildings*, *118*, 259–276. doi: 10.1016/j.enbuild.2016.02.024

Effets contradictoires/paradoxaux



Impacts négatifs (AUGMENTATION)

Economies d'échelle (loi de T. Wright)

Amélioration de l'efficacité

Mutualisation des services (iPhone)
Miniaturisation

Législation, normes, interdictions, obligations

Impacts positifs (REDUCTION)

Modèle basé sur la croissance, augmentation de la consommation (obsolescence programmée, globalisation)

Effets rebonds (paradoxe de Jevons)

Complexification et accélération des versions
Difficultés à réparer

Obsolescence, complexification, impacts multi-sectoriels globalisés

Introduction à l'éco-conception logicielle

Définitions, enjeux et leviers

Définition (générale) de l’éco-conception

- L’Agence de la transition écologique (ADEME) définit l’éco-conception comme « une démarche préventive qui se caractérise par la prise en compte de l'environnement lors de la phase de conception ou d'amélioration d'un produit. L'objectif de cette démarche est d'améliorer la qualité écologique du produit, c'est-à-dire réduire ses impacts négatifs sur l'environnement tout au long de son cycle de vie, tout en conservant sa qualité d'usage ».
- En d’autres termes, éco-concevoir c’est chercher à réduire la quantité des ressources informatiques utilisées sans renoncer à l’utilité des services rendus et aux bénéfices associés.
- Le numérique responsable prend aussi en compte les aspects bénéfiques multi-sectoriels (y compris sur la société, la santé, la qualité de vie, l’éthique, etc.)

Eco-conception logicielle : les enjeux

- L'impact le plus fort est lié aux terminaux utilisateurs (PC, smartphones), et en particulier à la fabrication (70% eq CO2 en France)
- Les enjeux court-terme de l'éco-conception logicielle sont donc :
  - De minimiser les effets d'obsolescence et de favoriser la durabilité du matériel (donc minimiser les resources CPU, mémoire, disque, etc.)
  - De minimiser la consommation énergétique des terminaux (mais aussi des data centers et des réseaux, le tiers de l'énergie consommée)

%	Energie	GES	Eau	Ressources[1]
Utilisateurs	64 %	84 %	91 %	79 %
Réseau	21 %	10 %	5 %	15 %
Centres informatiques[2]	15 %	6 %	4 %	6 %

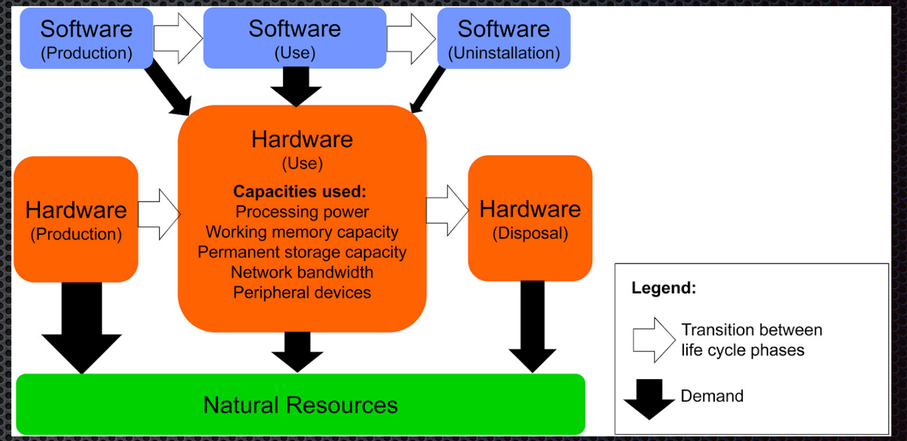
%	Energie	GES	Eau	Ressources[1]
Fabrication	41 %	83 %	88 %	100 %
Utilisation	59 %	17 %	12 %	0 %

Répartition des impacts du numérique en France

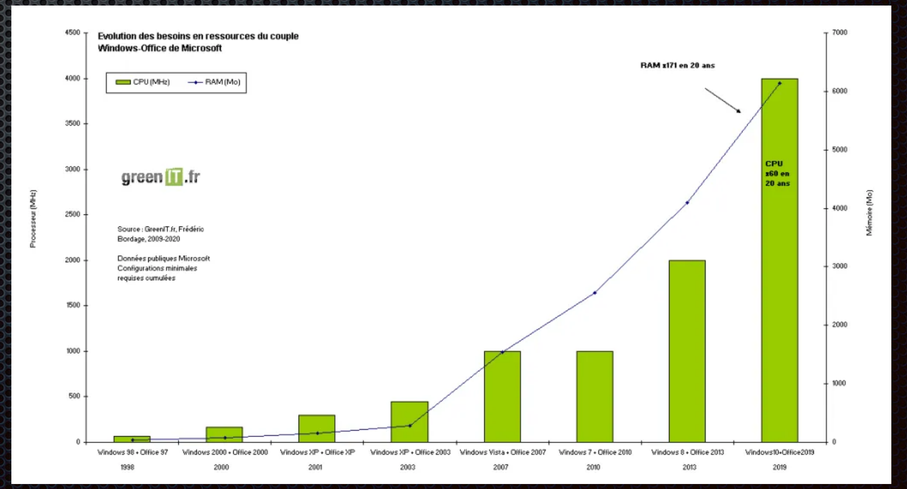
https://www.greenit.fr/wp-content/uploads/2020/06/2020-06-iNum-etude-impacts-numerique-France-rapport.pdf

Des leviers importants à exploiter

- Le logiciel est en début de chaîne
- La loi de Wirth*, aussi connue sous le nom de “loi de compensation” nous donne un effet de levier important



Kern E, Hilty LM, Guldner A, Maksimov YV, Filler A, Gröger J, Naumann S. Sustainable software products—Towards assessment criteria for resource and energy efficiency. Future Generation Computer Systems. 2018 Sep 1;86:199-210.



* “Slowing down programs is much faster than speeding up computers” (Niklaus Wirth)

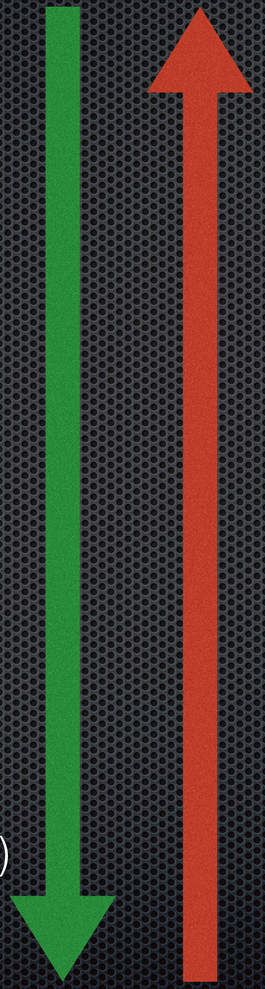
Exemples

- Optimiser en fonction de l'usage (statistiques)
  - Exemple : l'étude de l'usage du moteur de recherche Bing a permis par exemple, de mettre en avant que 90% des personnes n'étudient que le top 20 des liens retournés. L'exploitation de cette statistique permet de réduire l'impact environnemental des serveurs de 80 % (*).
- Segmenter des données en fonction des usages
  - « La Deutsche Bahn démontré qu'il était possible de diviser par 1350 la quantité de ressources informatiques nécessaires pour trouver l'horaire d'un train, si on utilise uniquement les données nécessaires » (source : Frédéric Bordage, GreenIT (**)).
- Utiliser des outils de compilation / transpilation / analyse de code, etc.
  - Par exemple, en 2010, Facebook a divisé par deux le nombre de serveurs nécessaires à son fonctionnement en modifiant le code de ses services (compilation du code PHP de son site en C++). Le réseau social émet ainsi 2 fois moins de GES qu'auparavant et a évité la construction d'un nouveau data center qui lui aurait coûté environ 100 millions de dollars et aurait émis des dizaines de tonnes de GES inutilement.

* https://www.greenit.fr/2014/12/03/bing-20-des-resultats-sollicitent-80-des-ressources/

** http://videos.senat.fr/Datas/senat/portail/video.1504468_5e2f71cae6b58.table-ronde-relative-a-l-empreinte-carbone-du-numerique

Effets paradoxaux en conception logicielle



Impacts positifs (REDUCTION)

Mutualisation
(base de données, serveurs, infrastructures)

Architecture, segmentation,
spécialisation (micro services)

Méthodologies, DDD, agilité

Optimisation, modernisation
(langages, compilation, protocoles,
gestionnaires de dépendances)

Dématérialisation

Obligations légales (GDPR)

Impacts négatifs (AUGMENTATION)

Complexification fonctionnelle (fatwares)

Complexification technique
(tests, CI/CD, virtualisation)

Frameworks, accélération des versions,
obsolescence, hype-oriented software engineering

Effets rebonds

Re-matérialisation
Stockage
Sécurité

Obligations légales (complexification, réécriture, traçabilité)

Efficacité énergétique

Mesure et axes d’optimisation

Efficacité v.s. Performance

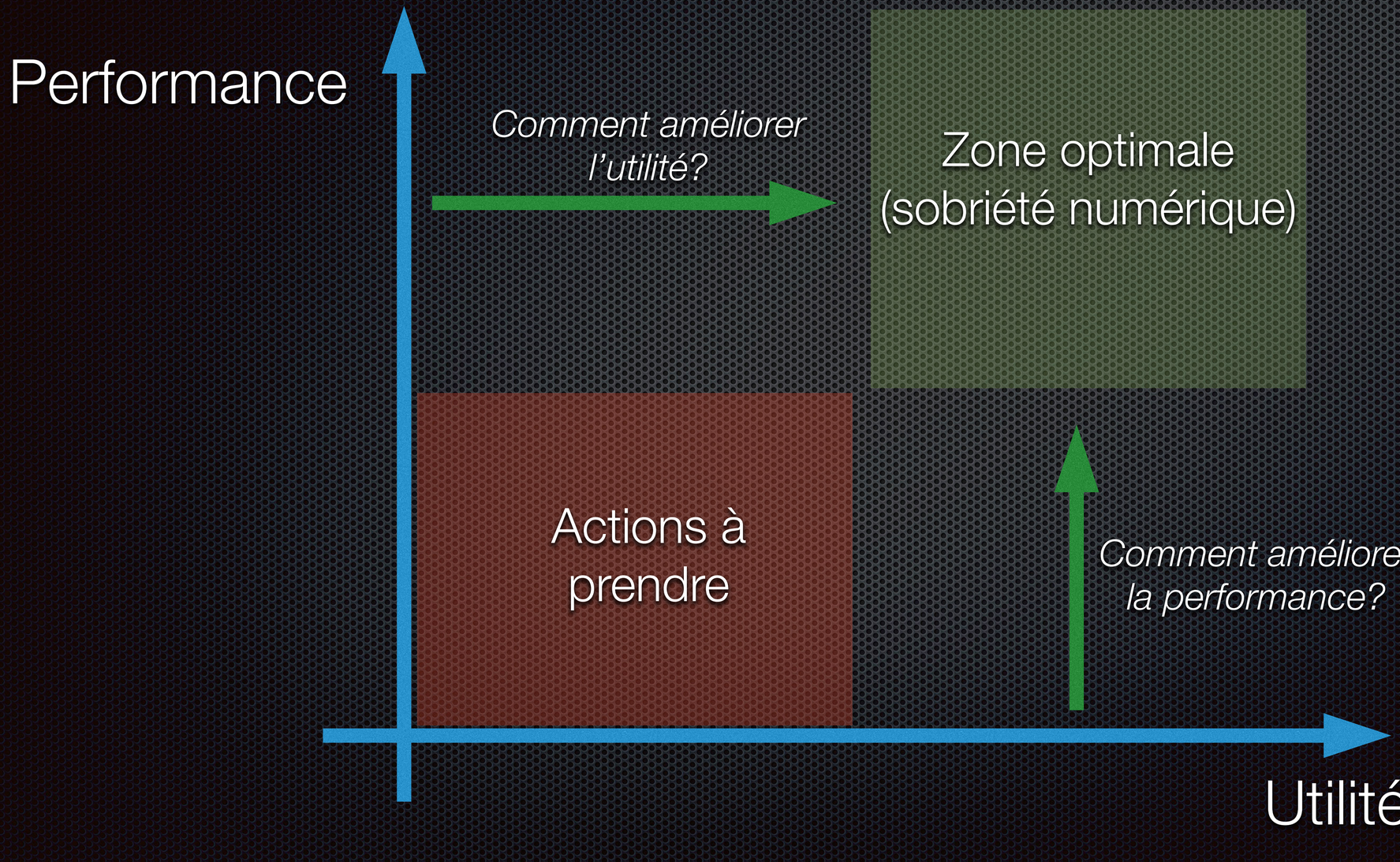
- La **performance** consiste à faire vite les tâches
- L'**efficacité** consiste à faire en priorité les tâches à forte valeur ajoutée
  - Exemple: répondre à 100 emails par jour v.s. répondre aux deux emails qui vont débloquer des situations critiques et vont me permettre de travailler sur d'autres sujets plus rentables
- L'efficacité est donc liée à un critère d'**utilité**
- Pour faire de la conception éco-responsable; il faut donc prendre en compte
  1. La consommation en énergie
  2. L'utilité **(sous contraintes QoS)** du service rendu

Efficacité Energétique = Utilité du service / Énergie utilisée

Sur la notion d’utilité

- La notion d’utilité est une notion systémique et relative
- Elle ne peut être définie que par une analyse transverse de l’éco-système dans une logique multi-critères, en évaluant les impacts croisés et en anticipant les potentiels effet rebonds
- L’utilité peut être associée à la durabilité et la qualité d’un service
- L’utilité peut être matérialisée dans le cahier des charges sous forme de contraintes de QoS mais nécessite une approche itérative pour être bien intégrée

Démarche itérative



Service par service
ET globalement
-
Itérations

Exemple

- Serveur de génération de rapports
- Contrainte de QoS : chaque rapport doit être généré en moins d'une seconde
- 15 vs 25 utilisateurs simultanés pendant 1 heure d'utilisation
- Résultat : 25% plus efficace avec 25 utilisateurs

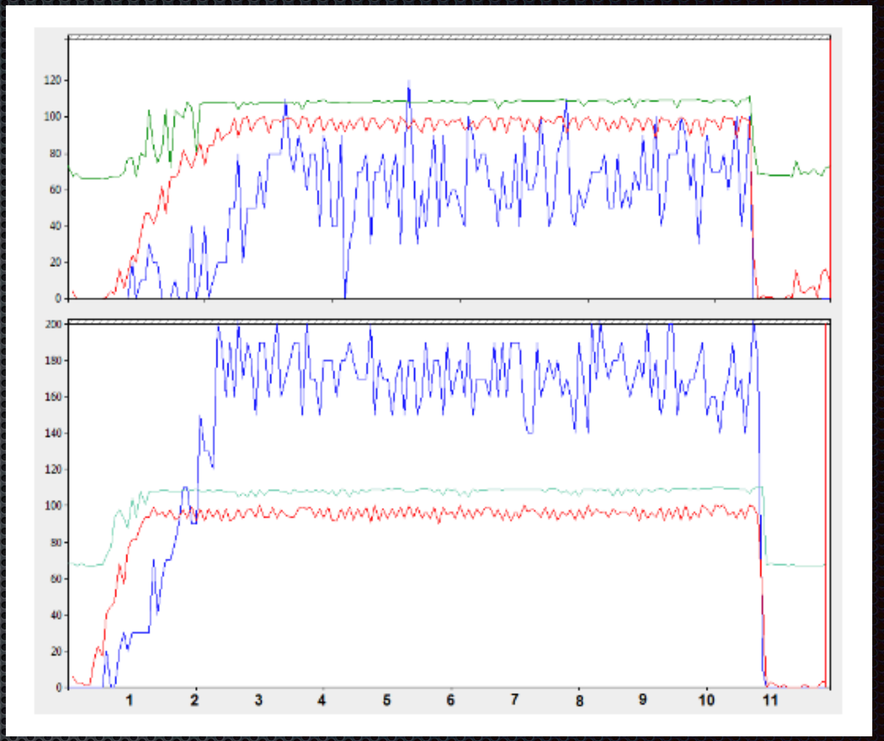


Table I
ENERGY PER USER

User Count	15 Users	25 Users
Energy Consumption	$7,3 \frac{Joules}{DeliveredChart}$	$5.5 \frac{Joules}{DeliveredChart}$
Average Response Time	0.5 seconds	0.8 seconds

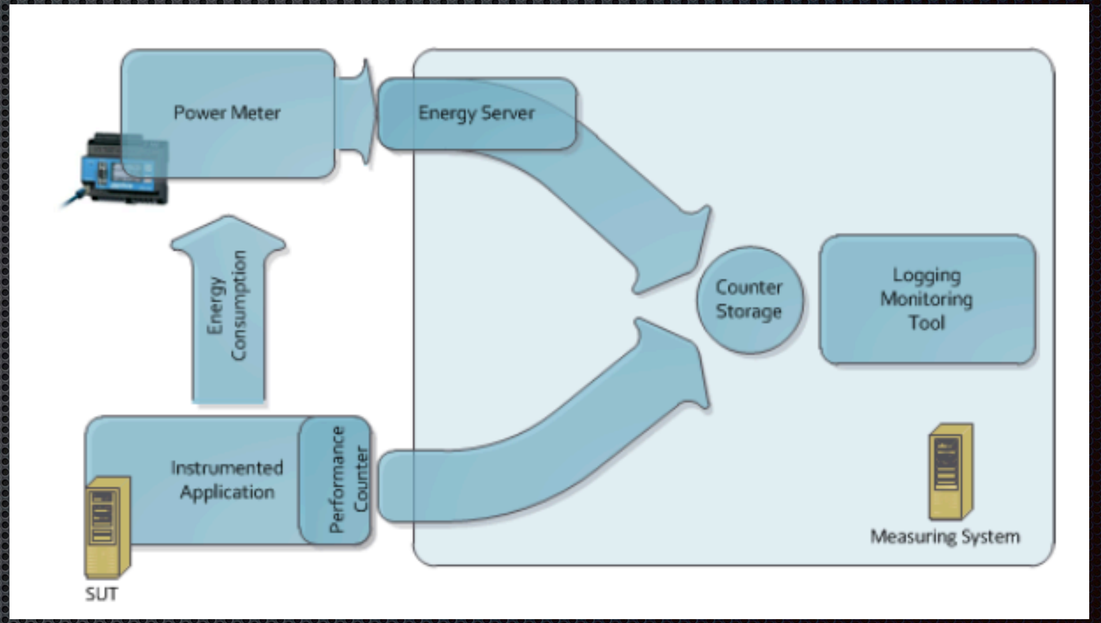
Source: How to Measure Energy-Efficiency of Software: Metrics and Measurement Results
Timo Johann, Markus Dick, Stefan Naumann, Eva Kern
Trier University of Applied Sciences, Germany

Mesure de l’énergie utilisée

- Avec un wattmètre
- Avec des sondes logicielles
  - PowerAPI (INRIA) - http://powerapi.org/
  - MC2 (CINCHEO) https://cincheo.com/2021/06/11/mc2-a-tool-to-remotely-monitor-computer-resources/
  - cloudcarbonfootprint.org
  - https://greenmetrics.io/

Monitoring et logging

- Les données de mesure de l'énergie doivent être corrélées au tâches effectuées (durée et performance d'exécution)
- Journaux systèmes et applications (outils de gestion de logs type Graylog), outils de monitoring type Zabbix
- Instrumentation de code pour augmenter la granularité des mesures (par exemple pour chaque fonction d'un programme ou d'une librairie)



La maîtrise de la complexité et des dépendances logicielles

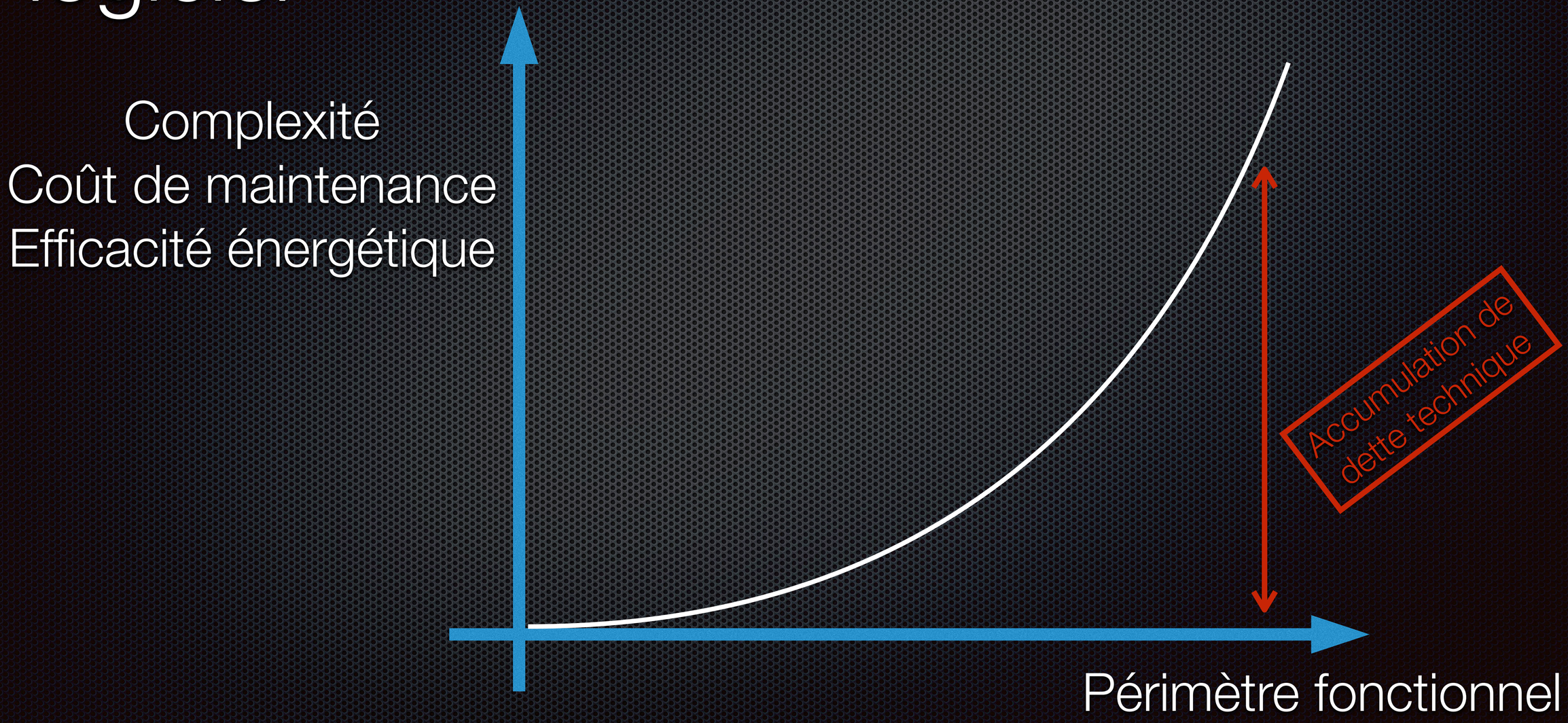
Le problème de l'abstraction et des dépendances

Origines de la complexité

- Loi de Moore + loi de Niklaus Wirth (voir “A Plea for Lean Software”)
  - Si on peut repousser l’optimisation et la simplification => on le fera
  - Si on peut appliquer une méthode plus énergivore pour gagner en productivité, en qualité, ou en sécurité => on le fera (cf, l’IA ou la blockchain)
- Economie = danger ancré dans les mentalités (crash de Ariane 5, bug de l’an 2000, big data (No SQL) et IA, …) - *ne pas confondre avec l’optimisation prématurée*
- De plus, l’éco-système logiciel est extrêmement complexe car les logiciels évoluent sans arrêt :
  - Corrections de bugs, Introduction de nouvelles features, Patch de sécurité
  - Evolution des besoins métiers et besoins de travail collaboratif (droit d’accès, sécurité, accès distant à l’information, etc.)
  - Optimisations (souvent bien) vs hype (souvent pas bien)
  - Montée en version des dépendances
  - Dépréciation des librairies et environnements d’exécution (exemples : JQuery (évolution des API core browser, animations), Java IO / NIO (évolution des supports physiques))…
  - Erosion d’architecture (inertie de la conception, dette technique, “crosscutting”…)
  - Evolution des outils de développement (voir https://www.fredrikholmqvist.com/pages/why-i-hate-frameworks.html)

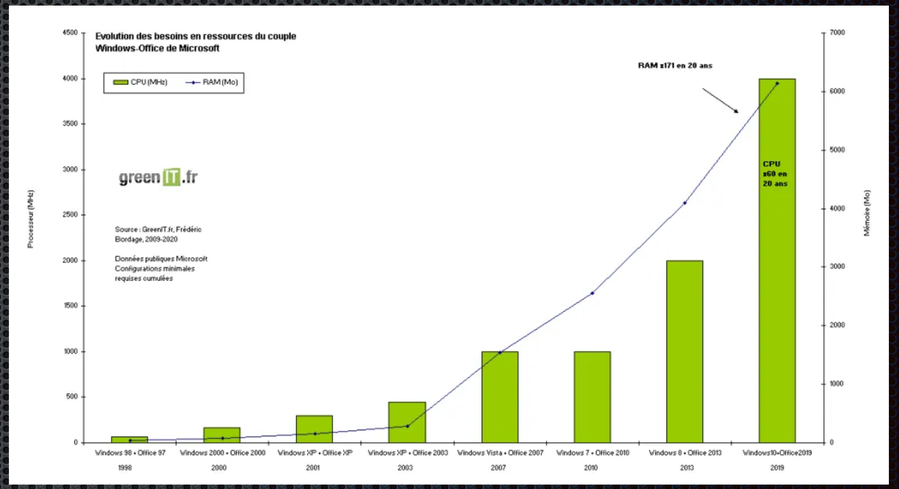
N. Wirth, "A Plea for Lean Software" in Computer, vol. 28, no. 02, pp. 64-68, 1995.
doi: 10.1109/2.348001
url: https://doi.ieeecomputersociety.org/10.1109/2.348001

Règle générale pour un composant logiciel



Stratégies pour maîtriser la complexité

- Compatibilité descendante / ascendante => abstraction
  - Fatware
- Imcompatibilité de versions, dépréciation
  - Semantic versioning Major.Minor.Patch (exemple Vue 2.2.1)
  - Modernisation, réécriture
- Compilation (langages)
- Gestionnaires de dépendances
- Architecture et conception



Importance des langages

Table 4. Normalized global results for Energy, Time, and Memory

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

https://greenlab.di.uminho.pt/wp-content/uploads/2017/10/sleFinal.pdf

Mais aussi…

- Importance des benchmarks
- Importance des usages
- Evolution : voir LLVM

- Impact des options :
  https://bugs.python.org/issue38980
  https://stackoverflow.com/questions/69503317/bubble-sort-slower-with-o3-than-o2-with-gcc

- Projet en cours sur l'optimisation de CPython :
  https://www.infoworld.com/article/3637073/python-stands-to-lose-its-gil-and-gain-a-lot-of-speed.html

Frameworks, hype et dépendances

- Hype 2007: GWT
- Hype 2012: AngularJS
- ES6 + dependency manager / npm
- Hype 2015: Reactjs, Vuejs, Angular 2 (+ TypeScript)
- Hype 2020 : framework less - http://vanilla-js.com/ - shadow DOM less frameworks
- Next hypes : No dependency manager? Low code?

Retrieve DOM element by ID

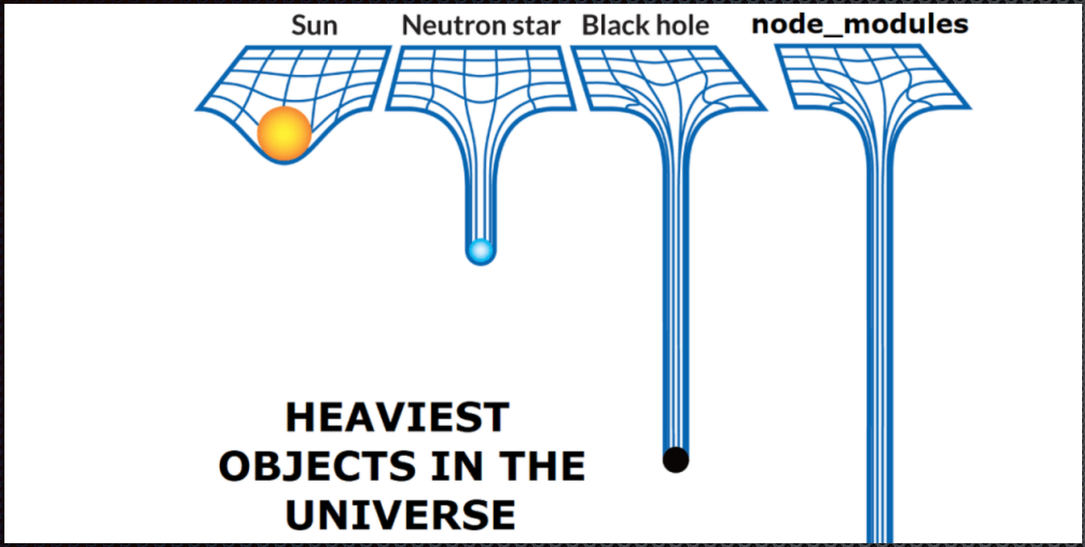
	Code	ops / sec
Vanilla JS	`document.getElementById('test-table');`	12,137,211
Dojo	`dojo.byId('test-table');`	5,443,343
Prototype JS	`$('test-table')`	2,940,734
Ext JS	`delete Ext.elCache['test-table']; Ext.get('test-table');`	997,562
jQuery	`$jq('#test-table');`	350,557
YUI	`YAHOO.util.Dom.get('test-table');`	326,534
MooTools	`document.id('test-table');`	78,802

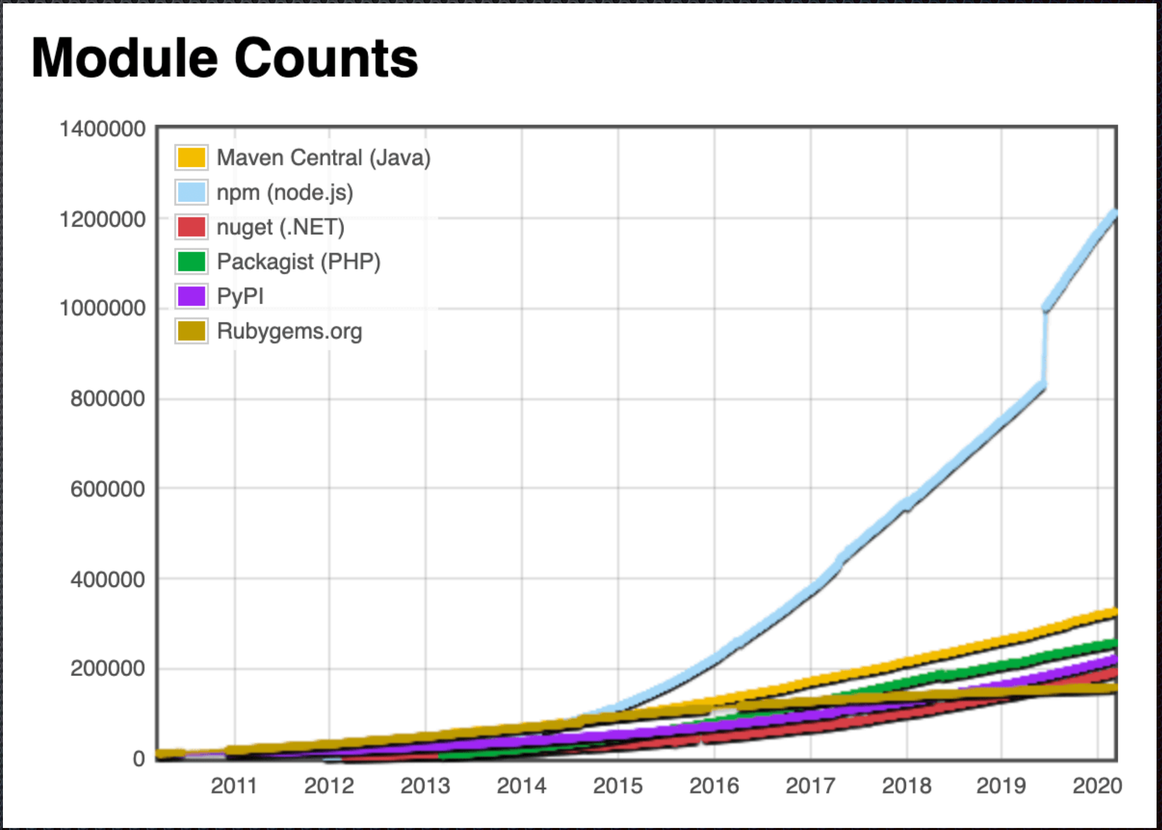
Retrieve DOM elements by tag name

	Code	ops / sec
Vanilla JS	`document.getElementsByTagName("span");`	8,280,893
Prototype JS	`Prototype.Selector.select('span', document);`	62,872
YUI	`YAHOO.util.Dom.getElementsBy(function(){return true;},'span');`	48,545
Ext JS	`Ext.query('span');`	46,915
jQuery	`$jq('span');`	19,449
Dojo	`dojo.query('span');`	10,335
MooTools	`Slick.search(document, 'span', new Elements);`	5,457

(voir https://www.fredrikholmqvist.com/pages/why-i-hate-frameworks.html)

Les gestionnaires de dépendances



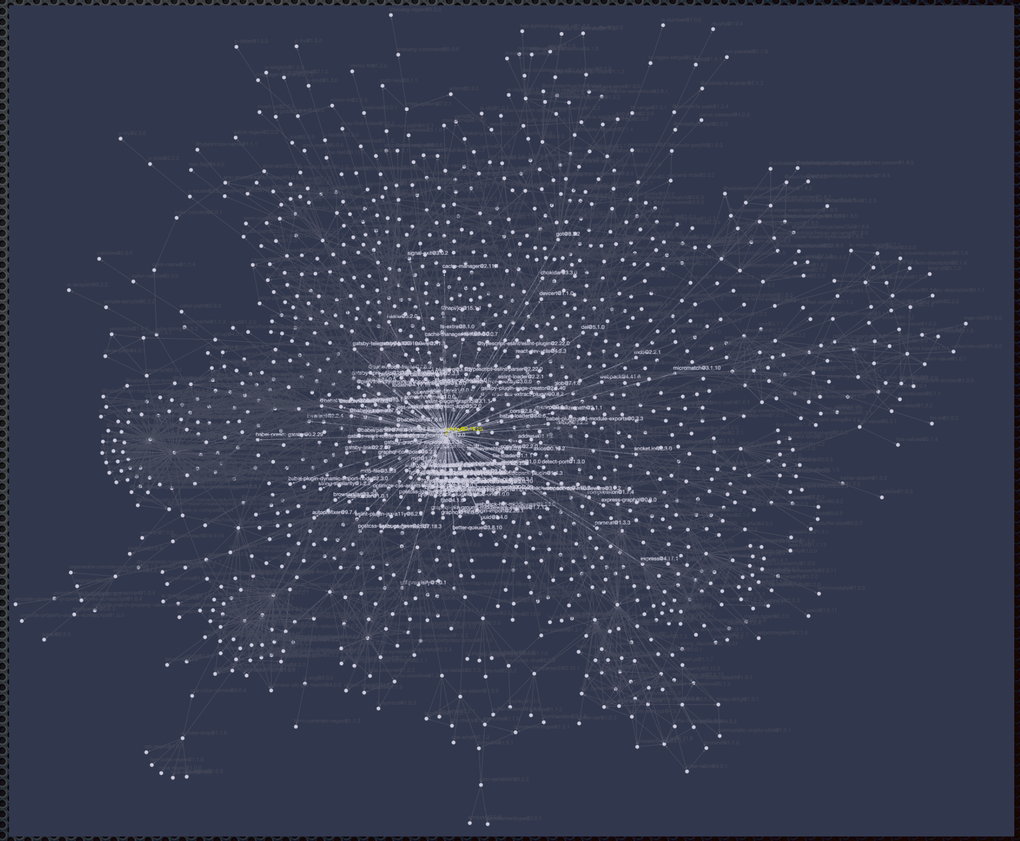


- Un cas d'école du paradoxe de Jevons !

The Javascript "ecosystem" is a hot mess and so is software development in general - https://nadh.in/blog/javascript-ecosystem-software-development-are-a-hot-mess/

Le cas du framework Gatsby

- Un framework front basé sur React.js
- 15K dépendances indirectes



https://npm.anvaka.com/#/view/2d/gatsby

Importance de l’architecture

Impact de l’architecture sur la complexité, la durabilité et l’utilisation des ressources

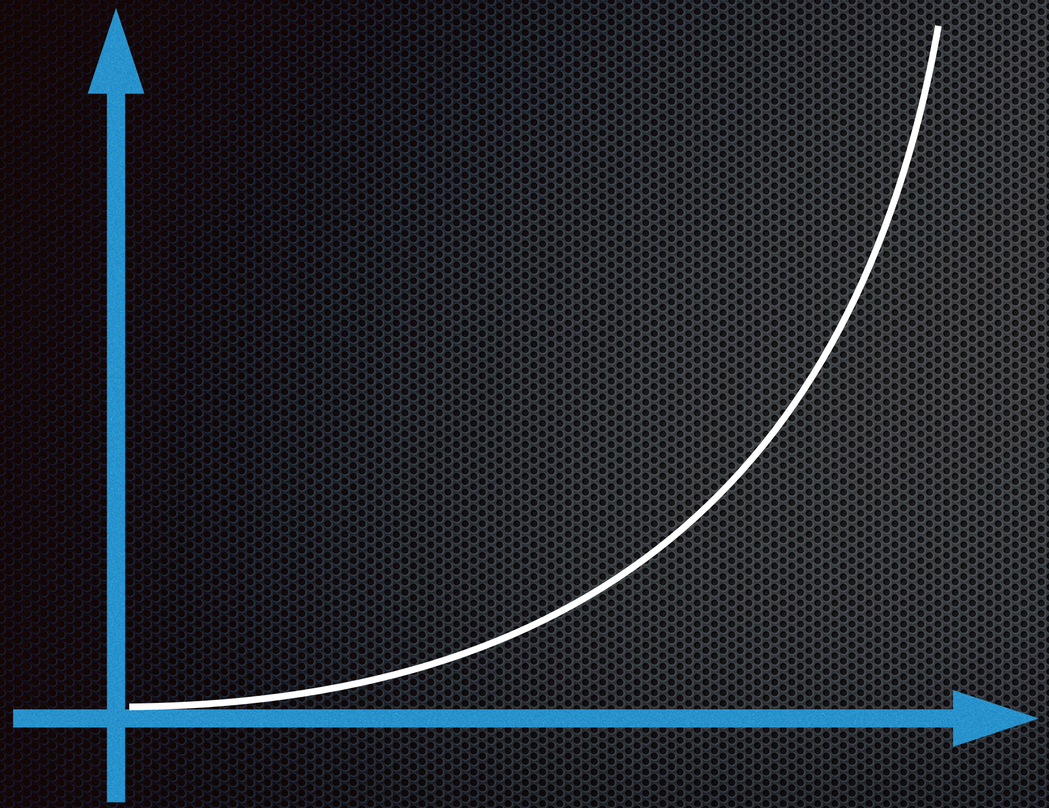
L’architecture est au coeur de la construction des logiciels

- Chaque architecture a des propriétés intrinsèques différentes en terme de durabilité, de performance, et d’utilité (Pyramides v.s. Cathédrales v.s. Chateaux Forts (Vauban), v.s. tour Eiffel, v.s. HLM)
  - Les conséquences négatives de choix architecturaux inappropriés augmentent exponentiellement avec le périmètre fonctionnel

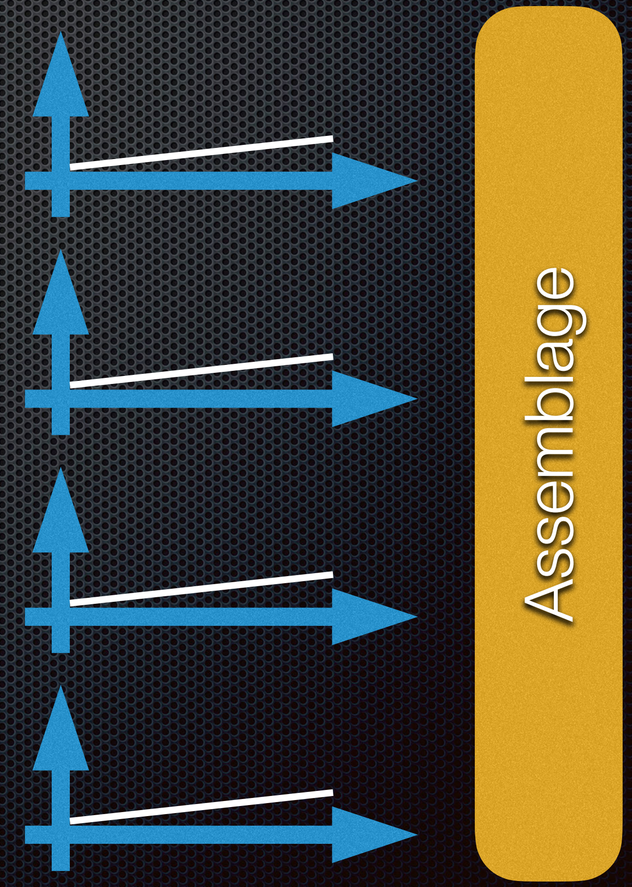
Architecture hexagonale et micro-services

- L'approche DDD et micro-services permet de mieux architecturer les logiciels pour réduire les impacts croisés et les effets de bords
  - Stabilisation du code métier, inversion des dépendances (architecture hexagonale)
  - Coût d'entrée élevé => mais a priori bénéfique sur le long terme (à confirmer)
- Par contre, la complexité fonctionnelle oblige à mettre en place des tests et des pipelines de CI/CD extrêmement coûteux (et induisant une complexité technique important)

L’approche par composants (micro-services, micro front ends)

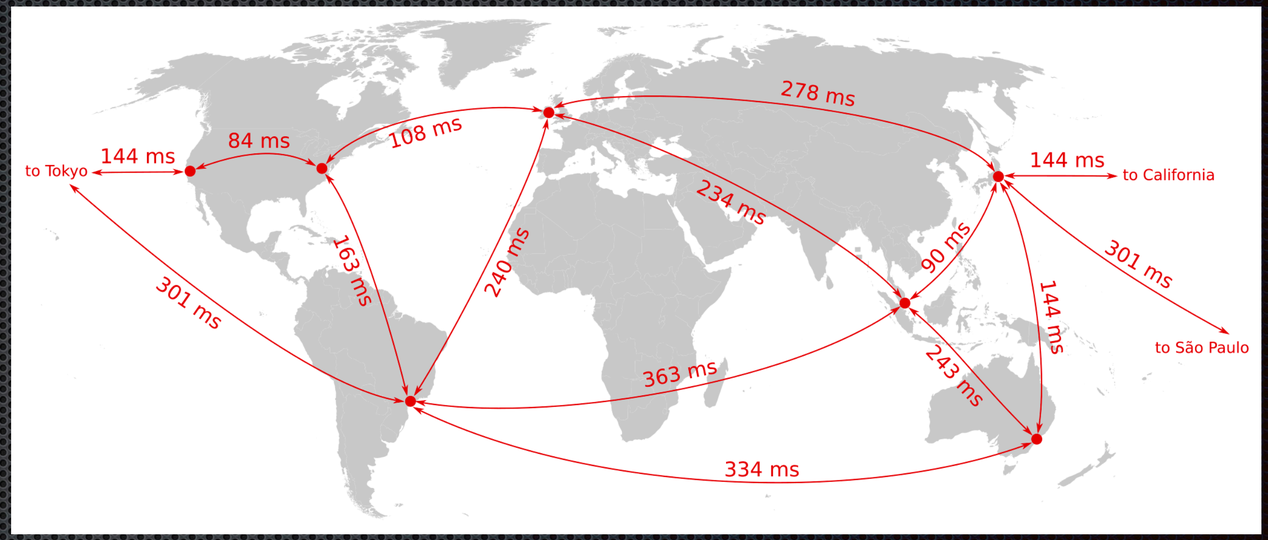


vs.



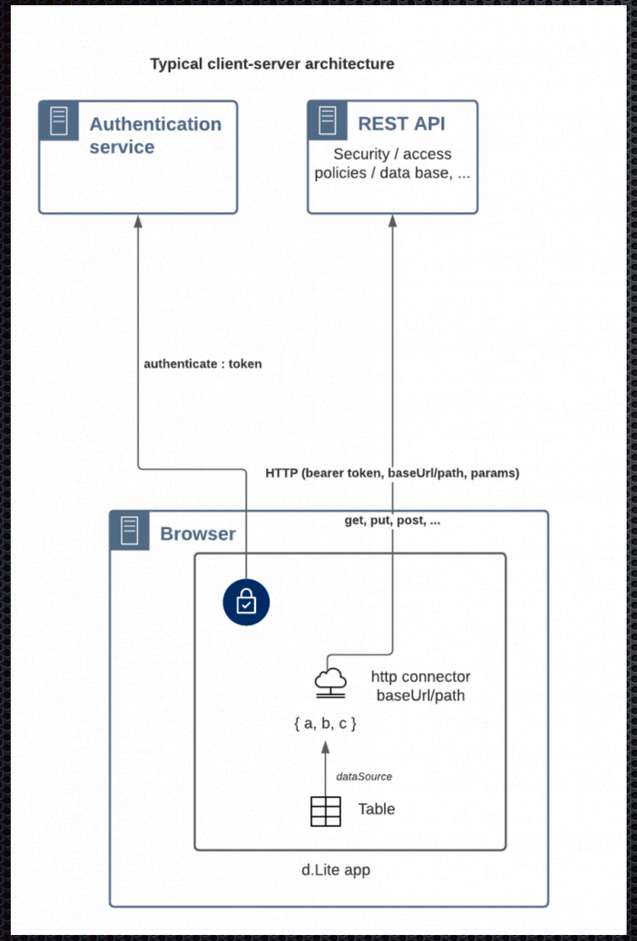
Qu’est-ce que le Local First (LF)?

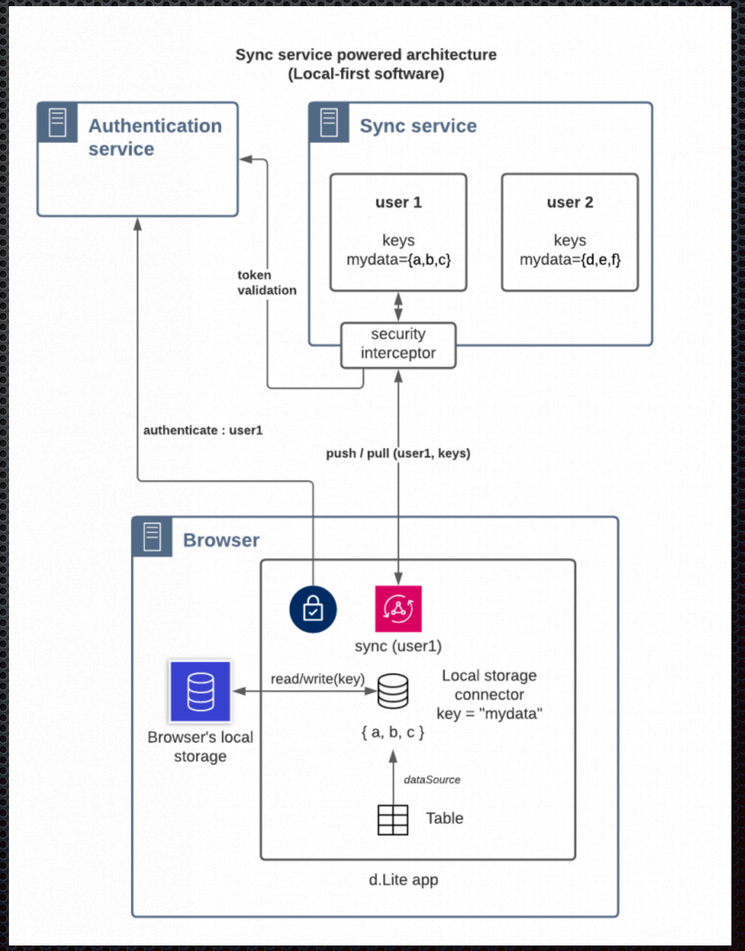
- Comme son nom l’indique, le LF consiste à :
  - Développer l’application en local, comme si elle était mono utilisateur
  - Puis, s’appuyer sur un protocole de **synchronisation** pour la partie multi-devices
  - Et un protocole de **sharing** pour les aspects collaboratifs
- Ainsi, chaque utilisateur reste maître de ses données (sécurité by design)



Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), October 2019, pages 154–178. doi:10.1145/3359591.3359737

L’architecture C/S vs. LF





Propriétés du LF

- Feedback instantané pour l'utilisateur, le réseau est optionnel => moins de resources et d'énergie consommée
- La donnée est distribuée sur les terminaux (sync protocol), ce qui assure une tolérance au panne "by design" (exemple: crash OVH ou panne Facebook) => donc moins de ressources nécessaire pour la construction et la sécurité des Data Centers
- Collaboratif "by design" (sharing protocol)
- La donnée vous appartient entièrement (de manière privée) et dure aussi longtemps que vous le souhaitez
- Ainsi, le LF est une possible voie pour construire des logiciels plus responsables
  - Protection des données, confiance et durabilité
  - Économie des stack backend spécifiques (et la CI/CD associée) de mieux tirer partie des ressources existantes des terminaux
  - Diminution de la complexité fonctionnelle et technique (logique "fatware" v.s. logique de différentiation en fonction des use cases et des communautés d'utilisateurs)

Conclusions et préconisations

Quelques conseils pour construire des logiciels plus responsables et plus durables

Prendre la mesure du challenge

- Problème complexe :
  - Beaucoup d'effets transverses, inattendus (loi de compensation) contradictoires et paradoxaux
  - Un enjeux à tous les niveaux (métier, politique, technique)
  - Guide de référence de la conception numérique responsable : https://gr491.isit-europe.org/ - *495 recommendations!!!*
- Les pièges à éviter :
  - Attention à l'électricité décarbonée
  - Attention aux référentiels et aux benchmarks => il faut mesurer et toujours mesurer
  - Attention au green washing et à l'évaluation par silo: exemple les trottinettes à Paris "Votre trajet est sans émission de CO2"
  - Attention à la peur de la "décroissance"

Comment faire ?

- Comment valoriser l'optimisation ?
  - UX, consommation énergétique, comparaison à la concurrence
  - Attention à la complexification, attention aux effets rebonds, aux compensations
- Comment valoriser l'utilité ?
  - Exemple des réseaux sociaux : obsolescence programmée de la data (c.f. la Civilisation du Poisson Rouge) v.s. nombre d'utilisateurs actifs (critère de valorisation basé sur la croissance)
  - Il faut revenir à une logique de qualité, de sobriété et de durabilité (v.s. une logique de volume) => c'est la "décroissance"
  - Il faut faire évoluer ses "valeurs": voir les 10 principes de Dieter Rams - https://ifworlddesignguide.com/design-specials/dieter-rams-10-principles-for-good-design
  - (logique basée sur la confiance)

Synthèse

1. Intégrer la sobriété dans les mentalités. Valoriser la sobriété numérique auprès de la direction, des clients et des collaborateurs. https://ifworlddesignguide.com/design-specials/dieter-rams-10-principles-for-good-design

2. Intégrer la sobriété et les impacts dans le cahier des charges, de la même façon qu'on intègre la GDPR ou la sécurité

3. Mettre en place une méthodologie adaptée : il s'agit d'un problème transverse qui touche à la fois aux usages et à la technique - avoir un process d'**idéation continue (car le logiciel et les usages évoluent et il faut sans arrêt se remettre en question) - faire du top down et du bottom-up.**

4. **Garder à l'esprit que l'architecture technique et la conception sont essentielles** pour améliorer l'efficacité et la durabilité des logiciels. Attention à l'*over-engineering* et les effets *hypes*. Voir le **local-first** et ses propriétés intrinsèques en terme de sécurité, de tolérance au pannes, etc.

5. Mettre en place des outils d'évaluation des impacts (référentiels) et mieux, des **outils de mesure et de benchmarking** (sinon cela revient à être aveugle sur les données "brutes" du terrain, qui sont celles qui comptent au final)