THESE de DOCTORAT DE
LYON 1 UNIVERSITE CLAUDE BERNARD

Ecole Doctorale 512
InfoMaths

Discipline : Informatique

Soutenue publiquement le 22/06/2026, par :

Edlira Nano

Obsolescence logicielle: analyse et stratégies de remédiation

---

Devant le jury composé de :

Aurélie Bugeau Professeure, Université de Bordeaux	**Rapporteuse**
Florence Maraninchi Professeure, Université Grenoble Alpes	**Rapporteuse**
Roberto Di Cosmo Professeur, Université Paris Cité, INRIA	**Examinateur**
Stéphanie Jean-Daubias Professeure, Lyon 1 Université Claude Bernard	**Examinatrice**
Sébastien Schulz Chargé de recherches, Centre Internet et Société, CNRS	**Examinateur**
Aurélien Tabard Maître de conférences, Lyon 1 Université Claude Bernard	**Directeur de thèse**
Nolwenn Maudet Maîtresse de conférences, Université de Strasbourg	**Invitée**

PHD THESIS of
LYON 1 UNIVERSITY CLAUDE BERNARD

Doctoral School 512
InfoMaths
Mathematics and Computer Science

In : Computer Science

Edlira Nano

Software Obsolescence: analysis and remediation strategies

---

under the supervision of
Aurélien Tabard

Jury composed by :

Aurélie Bugeau Professeure, Université de Bordeaux	**Rapporteur**
Florence Maraninchi Professeure, Université Grenoble Alpes	**Rapporteur**
Roberto Di Cosmo Professeur, Université Paris Cité, INRIA	**Examiner**
Stéphanie Jean-Daubias Professeure, Lyon 1 Université Claude Bernard	**Examiner**
Sébastien Schulz Chargé de recherches, Centre Internet et Société, CNRS	**Examiner**
Aurélien Tabard Maître de conférences, Lyon 1 Université Claude Bernard	**PhD supervisor**
Nolwenn Maudet Maîtresse de conférences, Université de Strasbourg	**Invited member**

Software Obsolescence: analysis and remediation strategies

Edlira Nano

Abstract

The aim of this doctoral dissertation is to analyze software obsolescence and existing remediation strategies, in light of the significant environmental impact of digital technologies, particularly that of electronic devices. To that end, I study two software ecosystems, Android and Debian, by using a qualitative research approach.

The first case study focuses on the Android ecosystem, the world's most widely used operating system (OS), where devices are often replaced and rarely updated more than two years after their release. I investigate what hinders Android development and maintenance, by conducting 12 interviews with key players in the ecosystem, supplemented by conference ethnography and analysis of technical literature. The analysis shows that the way code flows are organized across the various ecosystem actors inhibits updates, and outlines how these actors locate their maintenance efforts in different places to serve their strategic interests. The lack of updates appears at the kernel level, i.e., at the core of Android builds, as the code from phone vendors and system on chip manufacturers increasingly diverges from the original Linux kernel code. I show that Google, the main actor governing the ecosystem, addresses maintenance issues by shifting responsibility towards phone vendors. However, as vendors are the least inclined actors to maintain their code, the problem persists, leading to premature end-of-life for devices and, consequently, their obsolescence. I analyze how, driven by a concern for longevity, some vendors and alternative free open source mobile actors implement remediation strategies to maintain devices.

The second case study focuses on maintenance within Debian, a widely used free and open source operating system, based on the Linux kernel, maintained by a large international volunteer community. I investigate maintenance in Debian, by use of a community ethnographic approach during gatherings. I conducted 11 semi-structured interviews, supplemented by informal discussions, participant observation, analysis of online community tools and technical literature. The data were analysed using thematic analysis. I show that Debian's technical and social structure is designed to support and facilitate maintenance. This maintenance takes place at various levels: technical maintenance (code, tools), community maintenance (social and human interactions), work organisational maintenance (work processes, teams, roles) and infrastructuring (collective maintenance of inner infrastructure). At each of these levels, I analyse how maintenance in Debian is not only a technical process, but involves social interactions within and outside of the community. I also look at what helps maintenance and what hinders it.

Code in Debian is organized in packages. The analysis shows that the socio-technical relationship between Debian developers and external upstream developers plays an essential role in both the Debian package maintenance, and the upstream software maintenance. I highlight the crucial role that the collective process of developing, maintaining and improving the technical and social infrastructure in Debian plays in maintaining the operating system. I examine The Long Term Support program in Debian—offering 5 years of support on each release—highlighting an original business

model that the community has established, providing it with the revenue needed for maintenance while limiting corporate influence on maintenance decisions. Finally, the study identifies barriers in maintenance: human relations within the community: resolving conflicts, avoiding burnout, retaining members, attracting new ones seem to be important concerns for community members.

Reflecting on the conclusions drawn from the Android and Debian studies, I discuss the various strategies observed in terms of code flows between actors, and how they inhibit or facilitate maintenance. Breaking points can be technical: silent break of software support, lack of updates and upgrades, code obfuscation, proprietary or binary code, lack of documentation, hidden schematics, anti patterns in coding practices. This is particularly evident in the code provided by chip and smartphone vendors within the Android ecosystem. They can also be socio-economic, supported by technical and legal procedures, creating dynamics of power and abuse of power in software ecosystems. This is particularly true of Google in the Android ecosystem.

I discuss the play between openness and closure in software development and maintenance, as well as the importance of free and open source software practices, together with open standards in fostering code maintenance and building more independent and resilient systems. The study of these alternative FOSS ecosystems that succeed in offering longer software support and extend the lifespan of devices, shows that upstreaming and mainlining code are important maintenance strategies at the software development level. But it is social interactions within the communities developing the software and with external communities of users or developers, that are key to facilitate long term maintenance.

In examining maintenance practices within Debian, I emphasize that software maintenance is not limited to applying code updates, but is a socio-technical process that involves discussing, analyzing, and preparing these updates so that they align with the needs of the user and developer communities affected by them. The study on Debian shows that software updates do not necessarily result in hardware being discarded. As we further explore, obsolescence in the digital realm is made possible by business strategies and opportunities, through mutually reinforcing hardware and software techniques. Through planning, software enclosure, pseudo-history, and lobbying, obsolescence has been chosen and imposed, and lies at the heart of digital capitalism.

This dissertation concludes with a series of recommendations concerning, on the one hand, coding practices and, on the other hand, some regulatory measures necessary to ensure software maintenance while prohibiting the abuse of dominant positions. Furthermore, it seems essential to implement public policies aimed at supporting and fostering fundamental software ecosystems such as operating systems.

Keywords: obsolescence, digital obsolescence, software obsolescence, operating system, OS, software maintenance, Debian, Android, software ecosystems, socio-technical ecosystems, free and open source software, FOSS, digital capitalism.

Obsolescence logicielle : analyse et stratégies de remédiation

Edlira Nano

Résumé

Cette thèse de doctorat vise à analyser l'obsolescence logicielle ainsi que des stratégies de remédiation, dans le contexte de l'impact environnemental des technologies numériques, en particulier celui des appareils électroniques. À cette fin, j'étudie deux écosystèmes logiciels, Android et Debian, avec une approche de recherche qualitative.

La première étude de cas porte sur l'écosystème Android, le système d'exploitation le plus utilisé au monde, où les appareils sont souvent remplacés et rarement mis à jour plus de deux ans après leur sortie. J'ai étudié les obstacles au développement et à la maintenance d'Android à travers 12 entretiens avec des acteurs clés de l'écosystème, complétés par une étude ethnographique lors de conférences et une analyse de la littérature technique. L'analyse montre que la manière dont les flux de code sont organisés entre les différents acteurs de l'écosystème freine les mises à jour, et met en évidence comment ces acteurs concentrent leurs efforts de maintenance à différents niveaux pour servir leurs intérêts stratégiques. L'absence de mises à jour se manifeste au niveau du noyau, c'est-à-dire au cœur des versions d'Android, car le code des fabricants de téléphones et des fabricants de systèmes sur puce s'écarte de plus en plus du code original du noyau Linux. Je montre que Google, l'acteur principal régissant l'écosystème, aborde les problèmes de maintenance en transférant la responsabilité vers les fabricants de téléphones. Cependant, comme les fabricants sont les acteurs les moins enclins à maintenir leur code, le problème persiste, entraînant une fin de vie prématurée des appareils et, par conséquent, leur obsolescence. J'analyse comment, motivés par un souci de longévité, certains fabricants et acteurs alternatifs du secteur mobile libre et open source mettent en œuvre des stratégies de remédiation pour maintenir les appareils.

La deuxième étude de cas porte sur la maintenance au sein de Debian, un système d'exploitation libre et open source répandu, basé sur le noyau Linux et maintenu par une vaste communauté internationale de bénévoles. J'étudie la maintenance dans Debian, en recourant à une approche ethnographique communautaire lors d'évènements. J'ai mené 11 entretiens semi-structurés, complétés par des discussions informelles, une observation participante, l'analyse des outils communautaires en ligne et de la littérature technique. J'ai ensuite effectué une analyse thématique des données. Je montre que la structure technique et sociale de Debian est conçue pour soutenir et faciliter la maintenance. Cette maintenance s'effectue à différents niveaux : maintenance technique (code, outils), maintenance communautaire (interactions sociales et humaines), maintenance organisationnelle du travail (processus de travail, équipes, rôles) et maintenance infrastructurelle (maintenance collective de l'infrastructure interne). À chacun de ces niveaux, j'analyse comment la maintenance n'y est pas seulement un processus technique, mais implique des interactions sociales au sein et en dehors de la communauté. Je m'intéresse également à ce qui favorise l'entretien et à ce qui le freine.

Le code dans Debian est organisé en paquets. L'analyse montre que la relation socio-technique entre les développeurs Debian et les développeurs des outils externes en amont (upstream), joue un rôle essentiel tant dans la maintenance des paquets Debian que dans la maintenance des logiciels en amont. Je souligne le rôle crucial que joue le processus collectif de développement, de maintenance et d'amélioration de l'infrastructure technique et sociale de Debian dans la maintenance du système d'exploitation. J'examine le programme Long Time Support (LTS) garantissant cinq ans de sup-

port pour chaque version courante du système d'exploitation. Ce programme contient un modèle économique original mis en place par la communauté, qui procure au projet les revenus nécessaires à la maintenance, tout en limitant l'influence des entreprises sur les décisions internes de maintenance.

Enfin, l'étude souligne des obstacles à la maintenance. La pérennité des relations humaines au sein de la communauté : résoudre les conflits, éviter l'épuisement, fidéliser les membres, en attirer de nouveaux, semblent être des préoccupations importantes pour les membres de la communauté.

En m'appuyant sur les conclusions tirées des études d'Android et Debian, j'analyse les différentes stratégies observées en matière de flux de code entre les acteurs, ainsi que la manière dont elles entravent ou facilitent la maintenance. Les points de rupture peuvent être d'ordre technique : interruption silencieuse du support logiciel, absence de mises à jour et de mises à niveau, obfuscation du code, code binaire ou propriétaire, manque de documentation, schémas de fonctionnement non disponibles, anti-patterns dans les pratiques de codage. Cela est particulièrement évident dans le code fourni par les fabricants de puces et de smartphones au sein de l'écosystème Android. Les points de rupture peuvent également être socio-économiques, soutenus par des procédés techniques et juridiques, créant des dynamiques de pouvoir et agissant comme des instruments d'abus de pouvoir dans les écosystèmes logiciels. C'est particulièrement vrai pour Google dans l'écosystème Android.

J'aborde l'interaction entre ouverture et fermeture du code dans le développement et la maintenance des logiciels, ainsi que l'importance des pratiques en matière de logiciels libres et open source, associées aux standards ouverts, pour favoriser la maintenance du code et construire des systèmes plus indépendants et résilients. L'étude des écosystèmes alternatifs libres et open source qui parviennent à offrir un support plus long et à prolonger la durée de vie des appareils, montre que les pratiques d'upstreaming de code et de mainlining du noyau Linux sont des stratégies de maintenance importantes au niveau du développement logiciel. Mais ce sont les interactions sociales, à la fois au sein des communautés développant le logiciel et avec les communautés externes d'utilisateurs ou de développeurs, qui sont essentielles pour faciliter la maintenance à long terme.

En examinant les pratiques de maintenance au sein de Debian, je souligne que la maintenance logicielle ne se limite pas à l'application de mises à jour du code, mais qu'il s'agit d'un processus socio-technique qui implique de discuter, d'analyser et de préparer ces mises à jour afin qu'elles répondent aux besoins des communautés d'utilisateurs et de développeurs concernés.

L'étude sur Debian montre que les mises à jour logicielles n'entraînent pas nécessairement l'obsolescence du matériel. L'obsolescence numérique est rendue possible grâce à des stratégies économiques et des opportunités, tantôt matérielles, tantôt logicielles, qui s'entremêlent et se renforcent mutuellement. À travers la planification, l'enclosure logicielle, la pseudo-histoire, le lobbying, l'obsolescence a été choisie et imposée, et se trouve placée au coeur du capitalisme numérique.

Cette thèse se termine par une série de recommandations concernant, d'une part, des pratiques de codage et, d'autre part, des mesures réglementaires nécessaires pour garantir la maintenance des logiciels tout en interdisant l'abus de position dominante. En outre, il semble essentiel de mettre en œuvre des politiques publiques visant à soutenir et à favoriser des écosystèmes logiciels fondamentaux tels que les systèmes d'exploitation.

Mots-clés: obsolescence logicielle, système d'exploitation, maintenance logicielle, impacts environnementaux du numérique, Debian, Android, logiciels libres, open source, capitalisme numérique.

Résumé substantiel

Obsolescence logicielle : analyse et stratégies de remédiation

Edlira Nano

Introduction

Cette thèse de doctorat vise à analyser l'obsolescence logicielle ainsi que des stratégies de remédiation, dans le contexte de l'impact environnemental des technologies numériques, en particulier celui des appareils électroniques. À cette fin, j'étudie deux écosystèmes logiciels, Android et Debian, avec une approche de recherche qualitative.

La production d'obsolescence logicielle: le cas de l'écosystème Android

La première étude de cas porte sur l'écosystème Android, le système d'exploitation le plus utilisé au monde, où les appareils sont souvent remplacés et rarement mis à jour plus de deux ans après leur sortie (chapitre 2).

J'ai étudié les obstacles au développement et à la maintenance d'Android à travers 12 entretiens avec des acteurs clés de l'écosystème, complétés par une étude ethnographique lors de conférences et une analyse de la littérature technique.

L'analyse montre que la manière dont les flux de code sont organisés entre les différents acteurs de l'écosystème freine les mises à jour, et met en évidence comment ces acteurs concentrent leurs efforts de maintenance à différents niveaux pour servir leurs intérêts stratégiques. L'absence de mises à jour se manifeste au niveau du noyau, c'est-à-dire au cœur des versions d'Android, car le code des fabricants de téléphones et des fabricants de systèmes sur puce s'écarte de plus en plus du code original du noyau Linux. Je montre que Google, l'acteur principal régissant l'écosystème, aborde les problèmes de maintenance en transférant la responsabilité vers les fabricants de téléphones. Cependant, comme les fabricants sont les acteurs les moins enclins à maintenir leur code, le problème persiste, entraînant une fin de vie prématurée des appareils et, par conséquent, leur obsolescence. J'analyse comment, motivés par un souci de longévité, certains fabricants et acteurs alternatifs du

secteur mobile libre et open source mettent en œuvre des stratégies de remédiation pour maintenir les appareils.

La maintenance dans le projet Debian

La deuxième étude de cas porte sur la maintenance au sein de Debian (chapitre 3), un système d'exploitation libre et open source répandu, basé sur le noyau Linux et maintenu par une vaste communauté internationale de bénévoles. J'étudie la maintenance dans Debian, en recourant à une approche ethnographique communautaire lors d'évènements. J'ai mené 11 entretiens semi-structurés, complétés par des discussions informelles, une observation participante, l'analyse des outils communautaires en ligne et de la littérature technique. J'ai ensuite effectué une analyse thématique des données. Je montre que la structure technique et sociale de Debian est conçue pour soutenir et faciliter la maintenance. Cette maintenance s'effectue à différents niveaux : maintenance technique (code, outils), maintenance communautaire (interactions sociales et humaines), maintenance organisationnelle du travail (processus de travail, équipes, rôles) et maintenance infrastructurelle (maintenance collective de l'infrastructure interne). À chacun de ces niveaux, j'analyse comment la maintenance n'y est pas seulement un processus technique, mais implique des interactions sociales au sein et en dehors de la communauté. Je m'intéresse également à ce qui favorise l'entretien et à ce qui le freine.

Le code dans Debian est organisé en paquets. L'analyse montre que la relation socio-technique entre les développeurs Debian et les développeurs des outils externes en amont (upstream), joue un rôle essentiel tant dans la maintenance des paquets Debian que dans la maintenance des logiciels en amont.

Je souligne le rôle crucial que joue le processus collectif de développement, de maintenance et d'amélioration de l'infrastructure technique et sociale de Debian dans la maintenance du système d'exploitation. J'examine le programme Long Time Support (LTS) garantissant cinq ans de support pour chaque version courante du système d'exploitation. Ce programme contient un modèle économique original mis en place par la communauté, qui procure au projet les revenus nécessaires à la maintenance, tout en limitant l'influence des entreprises sur les décisions internes de maintenance.

Enfin, l'étude souligne des obstacles à la maintenance. La pérennité des relations humaines au sein de la communauté : résoudre les conflits, éviter l'épuisement, fidéliser les membres, en attirer de nouveaux, semblent être des préoccupations importantes pour les membres de la communauté.

L'obsolescence, modèle économique du capitalisme numérique

Dans la suite de cette étude, nous interrogeons la vision de l'histoire du numérique, où le remplacement accéléré de chaque modèle par un nouveau est vu comme un progrès, et les définitions de l'obsolescence qui la sous-tendent. Loin d'être un phénomène technique, une «loi» naturelle gouvernant ce secteur, elle relève de la planification (comme l'admet l'expression anglaise originale « planned obsolescence », obsolescence planifiée).

Pour comprendre la manière dont l'obsolescence s'est imposée comme un modèle économique, une stratégie d'organisation de l'industrie informatique, on interroge ici le processus de miniaturisation du matériel, à travers notamment la «loi» de Moore et les feuilles de route qui mettent en

place, organisent, et généralisent des stratégies d'obsolescences de produits informatiques, et leur remplacement continu. S'ensuit une critique de l'informatique dans le nuage et les nouvelles opportunités d'obsolescence qu'elle offre et met en place. Enfin, nous explorons la dynamique historique de la privatisation des logiciels que nous proposons de nommer « enclosure logicielle » ainsi que les dynamiques modernes des usages informatiques telle la « nuagification » et ce que cela implique comme nouvelles obsolescences.

Discussion

En m'appuyant sur les conclusions tirées des études d'Android et Debian, j'analyse les différentes stratégies observées en matière de flux de code entre les acteurs, ainsi que la manière dont elles entravent ou facilitent la maintenance. Les points de rupture peuvent être d'ordre technique : interruption silencieuse du support logiciel, absence de mises à jour et de mises à niveau, obfuscation du code, code binaire ou propriétaire, manque de documentation, schémas de fonctionnement non disponibles, anti-patterns dans les pratiques de codage. Cela est particulièrement évident dans le code fourni par les fabricants de puces et de smartphones au sein de l'écosystème Android. Les points de rupture peuvent également être socio-économiques, soutenus par des procédés techniques et juridiques, créant des dynamiques de pouvoir et agissant comme des instruments d'abus de pouvoir dans les écosystèmes logiciels. C'est particulièrement vrai pour Google dans l'écosystème Android.

J'aborde l'interaction entre ouverture et fermeture du code dans le développement et la maintenance des logiciels, ainsi que l'importance des pratiques en matière de logiciels libres et open source, associées aux standards ouverts, pour favoriser la maintenance du code et construire des systèmes plus indépendants et résilients. L'étude des écosystèmes alternatifs libres et open source qui parviennent à offrir un support plus long et à prolonger la durée de vie des appareils, montre que les pratiques d'upstreaming de code et de mainlining du noyau Linux sont des stratégies de maintenance importantes au niveau du développement logiciel. Mais ce sont les interactions sociales, à la fois au sein des communautés développant le logiciel et avec les communautés externes d'utilisateurs ou de développeurs, qui sont essentielles pour faciliter la maintenance à long terme.

En examinant les pratiques de maintenance au sein de Debian, je souligne que la maintenance logicielle ne se limite pas à l'application de mises à jour du code, mais qu'il s'agit d'un processus socio-technique qui implique de discuter, d'analyser et de préparer ces mises à jour afin qu'elles répondent aux besoins des communautés d'utilisateurs et de développeurs concernés.

Conclusion

L'étude sur Debian montre que les mises à jour logicielles n'entraînent pas nécessairement l'obsolescence du matériel. Comme analysé dans le chapitre 4, l'obsolescence numérique est rendue possible grâce à des stratégies économiques et des opportunités, tantôt matérielles, tantôt logicielles, qui s'entremêlent et se renforcent mutuellement. À travers la planification, l'enclosure logicielle, la pseudo-histoire, le lobbying, l'obsolescence a été choisie et imposée, et se trouve placée au coeur du capitalisme numérique.

Cette thèse se termine par une série de recommandations concernant, d'une part, des pratiques de codage et, d'autre part, des mesures réglementaires nécessaires pour garantir la maintenance des

logiciels tout en interdisant l'abus de position dominante. En outre, il semble essentiel de mettre en œuvre des politiques publiques visant à soutenir et à favoriser des écosystèmes logiciels fondamentaux tels que les systèmes d'exploitation.

En nous appuyant sur notre analyse et les informations fournies par nos interlocuteurs, nous avons identifié un ensemble de pratiques susceptibles de faciliter la maintenance des logiciels:

- les mises à jour ne devraient pas rendre le matériel obsolète;
- les mises à jour ne doivent pas entraîner de modifications cassant la compatibilité et doivent garantir la rétrocompatibilité sur de longues périodes (par exemple, au moins 10 ans);
- si des modifications entraînant une incompatibilité devaient être apportées, les mises à jour devraient être mises de côté et faire l'objet de discussion et évaluation avec les communautés d'utilisateurs et de développeurs concernées, jusqu'à ce qu'une solution et un processus communs soient mis en place.

Étant donné que les pratiques de maintenance ne sont mises en œuvre que lorsqu'elles correspondent aux valeurs et aux objectifs des parties prenantes concernées, il est également nécessaire de les faire respecter par le biais de la réglementation. Les recommandations suivantes vont des mesures les plus simples à des transformations à plus grande échelle qui permettraient d'améliorer plus radicalement la longévité:

- exiger des fabricants d'appareils qu'ils publient des plans de mise à jour et de mise à niveau pour chaque appareil, et veiller à ce qu'ils soient respectés;
- exiger des fabricants d'appareils, des fabricants de puces et de composants ainsi que des éditeurs de logiciels qu'ils mettent à disposition l'ensemble du code, des schémas et de la documentation relatifs au fonctionnement du matériel dès lors qu'ils ne fournissent plus de mises à jour, et à l'expiration de la période de garantie et de support;
- intégrer les pratiques de upstreaming et mainlining dans les licences libres et open source;
- exiger que tous les composants d'un système d'exploitation, des chaînes de boot, ainsi que les micrologiciels fassent l'objet d'une maintenance pendant au moins 10 ans;
- tous les composants du système d'exploitation, des chaînes de boot et les micrologiciels devraient être tenus de respecter (ou de définir) des normes publiques ouvertes, afin d'éviter les dépendances vis-à-vis de logiciels propriétaires;
- considérer les systèmes d'exploitation, les micrologiciels et les logiciels de chaîne de boot comme des infrastructures publiques, gérées comme des biens communs numériques.

Publications

1. Edlira Nano, Léa Mosesso, Nolwenn Maudet, Aurélien Tabard, *Producing software obsolescence: the case of Android OS*. Submitted to **ACM Journal on Responsible Computing**, February 2026.

2. Edlira Nano et Jeanne Guien, *L'obsolescence, modèle économique du capitalisme numérique*. Book chapter in **Capitalisme numérique**, editorial direction: Olivier Alexandre, Benjamin Loveluck. C&F Éditions, to appear in 2026.

3. Léa Mosesso, Nolwenn Maudet, Edlira Nano, Thomas Thibault, Aurélien Tabard, *Obsolescence Paths: living with aging devices*. **ICT4S 2023 - International Conference on Information and Communications Technology for Sustainability**, 9 June 2023, Rennes (France). doi: 10.1109/ICT4S58814.2023.00011. HAL: hal-04097867.

4. Edlira Nano, *Digital Obsolescence*. Doctoral Symposium of **ICT4S 2023, International Conference on Information and Communications Technology for Sustainability**, 9 June 2023, Rennes (France), pp. 36-47. HAL: hal-04107104.

Abbreviations

AM : Application Manager (in Debian)

AOSP : Android Open Source Project

API : Application Programming Interface

COTS : Commercial Off The Shelf

CPU : Central Processing Unit

DD : Debian Developer

DFSG : Debian Free Software Guidelines

DIY : Do-It-Yourself

DM : Debian Maintainer

DRM : Digital Rights Management

DSC : Debian Social Contract

FLOSS : Free/Libre and Open-Source Software, same as FOSS

FOSS : Free and Open-Source Software

GM : General Motors

GNU GPL : GNU General Public Licence

ICT : Information and Communication Technology

ITRS : International Technology Roadmap for Semiconductors

IRDS : International Roadmap for Devices and Systems

OS : Operating System

PC : Personal Computer

SDK : System Development Kit

SoC : System on Chip

TSMC : Taiwan Semiconductor Manufacturing Company

WEEE : Waste Electric and Electronic Equipment

Glossary

- Backporting: the process of porting a software update that was developed for a current version of a software entity, to an older version of the software. It is a maintenance activity of the software development process, typically used for relatively small scope changes, such as fixing a software bug or security vulnerability, applying a *patch*.
- Downstream code: Also called downstream or the downstream. When code is derived from some other code, it is called a downstream code, while the original code from which the derivation takes place is called the upstream one. For example the Linux kernel is the upstream of the Android kernel, the Android kernel is a downstream code of the Linux kernel. Se also Upstream.
- Downstreaming: to downstream code is to take into account updates and changes that have occurred into the upstream code from which the downstream code derives, into the downstream code.
- Embedded software: software embedded in particular hardware, commonly known as embedded systems. Specialized for the particular hardware that it runs on and has time and memory constraints. Sometimes called firmware.
- Firmware: software embedded in particular hardware providing low-level control of this hardware. In simple embedded devices firmware may perform all the controls for the device, while in more complex devices, such smartphones or PCs, it provides the control of the device it is embedded in to higher-level software such as the OS.
- Forge: software forge are developer platforms, central place, that allows developers to create, store, manage, and share their code, documents, and files for a development project. Often coming with an online interface (GitHub, GitLab, Gitea, Codeberg etc), forges include code repositories and version control systems (often *git*) to organize code and allow collaboration. They also often include several project management and code development tools such as task management, testing, issue or bug tracking, code merging etc.
- Infrastructuring: the process of infrastructure development and the dynamic relationship between these infrastructures, the actors building, updating and using the infrastructure, and the environment where the infrastructure evolves. Examined in Science and Technology Studies (STS), in what is also called *Infrastructuring Studies*.
- Kernel: a computer program at the core of a the OS, that has complete control over everything in the system. It is the portion of the OS code that is always resident in memory and facilitates

interactions between hardware and software components. A full kernel controls all hardware resources via device drivers, arbitrates conflicts between processes concerning such resources, and optimizes the use of common resources, such as CPU, cache, file systems, etc., handles input/output requests from software, translating them into data-processing instructions for the central processing unit. See also the Linux kernel.

- Mainlining: In free and open source software development, *mainlining* is an important widespread coding and maintenance practice that refers to the integration of new developments into the main *mainline* source code branch in the project's repository. When developing a system derived from the Linux kernel, mainlining the Linux kernel means: (1) your system has to follow the mainline Linux kernel development (downstream its changes), and also (2) contribute back code to the Linux kernel as often as possible (upstreaming). See also Downstreaming and Upstreaming.

- Patch: a small modification of code applied to a your code version when you cannot mainline or upstream changes immediately. In maintenance of FOSS projects, the less patches a project has, the more simple its future maintenance will be. The idea is to apply changes in an upstream or mainline model, with the least patches possible. In this cases applying a patch is seen as a temporary solution for updating the code, before the upstreaming and mainlining process take place. Therefore, it is important when building the patch to prepare it for future upstream and mainline. Applying a patch is called *Patching*.

- Patching: Applying a patch. See also Patch.

- Porting: process of making an OS work for a specific hardware or device. For example, porting Debian to a new SoC hardware architecture such as an ARM SoC, or to a specific device such as an MNT reform computer. Other example: porting Android to a new phone means making the Android OS fully functional on this specific phone model, or developing an Android OS build for this phone.

- Rebasing: a development practice consisting in applying changes made to a derived code, to the original code from which the derivation occurs by resolving code conflicts, malfunctions, etc.

- Repository: code repository, a data structure that stores metadata for a set of files or directory structure, in order to store the code files and the history of changes made to them. Used on Version control systems. While the version control system is used to keep track of all the versions of a set of files, the repository keeps track of the files in the project.

- The Linux kernel: a free and open-source kernel created in 1991 that is used in many computer systems worldwide. It has been included in many FOSS operating system distributions, which are called Linux OSes or simply Linux. Debian is one of them. Android also uses the Linux kernel. The Linux kernel is also used in many embedded systems and IoT devices. See also Kernel.

- Upstream code: Also called the upstream, or upstream. When code is derived from some other code, the upstream is the original software or code from which the derivation takes place, while the derived code is a downstream code. For example the Linux kernel is the upstream of the

Android kernel. The upstream of the Emacs software package in Debian is Emacs. See also Downstream.

- Upstreaming: when updating code derived from an upstream project, upstreaming is the process of contributing back the new code to the upstream code. For example, when updating the Android kernel in order to take into account a new type of SoC, upstreaming consists in updating also the upstream Linux kernel.

- Version control system: in software development, a software allowing to manage several versions of source code or data for the project. It is often used to control source code in software developed collaboratively while allowing multiple developers to collaborate on a project. The first and most widespread version control system; *git* was developed to allow collaborative development of the Linux kernel system. Often included in a software forge, it uses code repositories.

Contents

List of Figures

List of Tables

Foreword

During my college years, in the early 2000s, I learned computer science surrounded by professors, researchers, and students who used Linux-based operating systems and other free and open-source software. These tools were affordable for both the university and the students, they were also packed with features that made it easy to work with hardware and software, view the code, modify it, break it, and reinstall it as many times as necessary to better understand and learn how hardware and software interact.

Through my software programming classes and activities, I learned how free software is not merely about open-source code and legal licences that guarantee the technical aspects of these principles, but that it is above all, a way of developing software within society and for society, about developing tools together, developers, users, researchers, to serve the common interest. By working collaboratively, building on the work of others, fostering improvement, adaptability, and diversity, the code could be modified, forked, or repurposed. I also learned how to train myself in this open, collaborative approach of developing software by also documenting my code, writing user documentation, using forges, version control systems, repositories and bug-tracking tools, that free and open source software were using long before the emergence of platforms such as Google Code or GitHub. I was also shown how to contribute back to free and open-source software that I used or considered important, by reporting or fixing bugs, writing missing documentation, or participating in the development.

In the proteomics research laboratory where I worked for five years as a software developer, our spectrometers—very expensive, massive and complex research instruments—could only be operated from a dedicated workstation located nearby, using specific proprietary software purchased from the supplier, in a very noisy and uncomfortable room, where simply wearing perfume could affect the spectrometer's calibration and, consequently, the analysis results. The data from the spectrometers came in a closed, proprietary and poorly documented format, and we all relied on the research community that had documented and standardized practices around this data format. The spectrometers themselves ran on proprietary software, that we were forced to trust, the source code not being available for verification or traceability. I learned how, in scientific research studies, it is important to be able to accurately trace the steps of the analysis and explain precisely how and what operations the algorithms performed, in order to ensure the reliability of the results, assess the margins of error, establish a reproducible process, and thereby confirm the integrity of the study. All of which, were impossible due to our spectrometer software restrictions. We had to blindly trust the vendor, the machine and their software.

I was genuinely impressed by my team's skills in handling the problems caused by this limited access to the software system of the spectrometer or by its opaque proprietary data format, and

how they imagined indirect ways to insure integrity of our analysis in these conditions. This made our lab's day-to-day work more difficult, though certainly exciting, but a far cry from the idea of software that I had experienced in my university training.

In the proteomics lab, my work consisted in developing a software analysing the data from the spectrometers, and adapting this analysis to multiple research needs and studies in our field. In practice, the first phase involved extracting the data from its proprietary format and converting it into a documented open format. Next, we implemented algorithms to clean, quantify, and analyze the data—the same work performed by our vendor's software on the spectrometer workstations—but this time, we could trace and explain each step of the analysis, ensure its reproducibility, allow for precise adjustment of analysis parameters to reflect the diversity of our proteomics experiments, while incorporating the latest research techniques in this field. Thus, one of the goals of my work was to free us from the constraints imposed by the spectrometer software we purchased, by developing our own software. We released this software under a copyleft free license, the GNU General Public License (GNU GPL), to ensure that all derivative code would remain open-source and continue to benefit the scientific community. My team and I also devoted a great deal of time into setting up tools for collaborative community use and development of our software. We also put effort in documenting and offering training for the use of the software. Users often told us about the software's scientific value in fine-tuning according to the experimental needs, and getting a traceable, reproducible treatment pipeline for the analysis.

Some years later, the company that manufactured and sold our spectrometers, offered to buy our software for an outrageous amount of money, in order to integrate it to their next generation spectrometers. Their condition was to stop distributing our software as a free and open source one, and give them the full ownership and distribution rights. I remember how my team and I discussed of the irony of this offer: if we agreed, our software would become the new vendor's software installed in the noisy room near the spectrometer, the very one we had been working so hard to replace by developing it. We refused the offer. But, we, scientists, users, developers, have experienced this situation move in the opposite direction many times, free and open-source software being developed, only to be acquired or unlawfully exploited and locked into proprietary systems—the very systems they were supposed to replace.

In my personal life, I also experienced similar usage restrictions and problems. I remember the CD-s I bought for my mother's birthday that contained Digital Rights Management software (DRM), preventing the CDs from being played by the stereo system that she had bought a year before from the same vendor. I remember my first smartphone, an Android Motorola, that had Facebook and Google search installed, constantly running, enforced on me, since I could not uninstall, deactivate or even remove them from my front screen, and how that led me to consider buying a new one sooner than I normally would have.

How is obsolescence related to these experiences? These software locks, lock-in mechanisms, licence restrictions, imposed terms of use, that come in a context of power dynamics from big digital companies, have influenced how I use or disuse these digital systems. They have led to discontinuities or disruptions in my use of the software or hardware, they became obsolete to me, but also to my friends, family or coworkers. At the same time, exploring hacking techniques, software, and alternative hardware solutions to avoid these problems, to have better control of digital systems, by engaging with communities that share this approach, has been an enriching experience for me. This experience has helped me rediscover my interest for computer science, and have shaped my

vision of computing systems that support, connect, and help build community, and that we govern collectively.

What ultimately led me to work on software obsolescence for this PhD thesis, was the desire to analyze it from a broader perspective than my own personal experience. By understanding obsolescence mechanisms, its various causes and manifestations in and through software, as well as ways to prevent, combat, remedy it, I am aiming to make a better sense of user experiences with technology, but also, of the significant impact of technology on the environment, our lives, and our society.

Chapter 1

Introduction

Contents

1.1 Research motivation: the unsustainability of the digital technology sector

The environmental crisis is one of the most important challenge of our society, as its causes and manifestations, such as climate change, the destruction of the biodiversity, the degradation of habitats

and natural resources, threaten the ecosystems we depend on as humans, and as a result our well-being and survival. Long overlooked, the impact of human activities and how our technological and socio-economic systems shape environmental sustainability is now the subject of significant debate. The framework of planetary boundaries [2, 3, 4] and of sustainable development have become a topic of public debates, discussed in the United Nations (UN) or the European Union (EU) sustainable development goals[1]. The right to a clean, healthy and sustainable environment has recently emerged as a universal human right (UN General Assembly Resolution A/76/L.75, 26 July 2022) [5]. Climate activism and climate justice initiatives have increased in number and intensity among civil society worlwide [6, 7].

The issue of the environmental footprint of digital technologies and related human activities has also been the subject of specific research in recent years. This research shows that the digital sector is experiencing steady and rapid growth, as is its environmental footprint, its direct and indirect impacts, despite targets set for a more sustainable technology [8, 9, 10, 11]. A new field of research has emerged on the environmental issues of computer science technology in a resource limited world as evidenced by international scientific gatherings such as the annual ICT4S (Information and Computer Science for Sustainability) conference or the Computing Within Limits workshop. Scholars have been questioning the way our field and institutions should deal with environmental and sustainability issues [12, 13, 14, 15]. The environmental issues have pushed many of them to call for a new direction in the research in computer science by questioning our own research practices [16] and topics. In this perspective Maraninchi suggests looking at the notion of 'anti-limits' [17], to revisit the way software is designed in computer science [18]. Scholars have called for a focus on *Undone Science*—topics of research and ways of practicing it that have not yet been investigated—by striving to consider social movements and civil society critique, and by taking an interdisciplinary perspective integrating ethics, science and technology studies, philosophy of science [19, 20, 21, 22]. From a policy and regulatory perspective, calls to analyze the environmental impacts of increasing digitization programs, and to take into account the planetary boundaries for sustainable development, have also emerged in recent years [13, 23, 24, 25, 26].

1.1.1 Device renewal: a non-exhaustive picture of the socio-environmental impacts of extraction and manufacturing

Device renewal accounts for a large part of the greenhouse gas emissions of ICT—or the carbon footprint—of information technology worldwide [27]. In France, the ADEME and Arcep study report states that as for 2022, consumer devices account for 50% of the digital carbon footprint, while data centers account for 46% [28, p.19]. A widespread methodology—also used in these reports—is the life-cycle assessment (LCA) of environmental effects through the life-cycle of the devices being assessed: from the extraction of raw materials required, the manufacturing or production process, the transport to the user emissions, the use or operational emissions to the end-of-life or disposal emissions, as illustrated in figure 1.1.

It is well admitted that the material extraction, manufacturing and end-of-life phases of devices (user devices or data center equipment) account for the majority of the emissions—in the ADEME & Arcep report they account for 60%, while the use phase for 40%. Therefore the device renewal issue is central to sustainability in the ICT sector.

[1] See the UN's and EU's sustainable development goals, last accessed in April 2026.

Figure 1.1: Material life cycle, The National Institute of Standards and Technology (NIST), 2011, Wikimedia Commons, public domain.

But, the impacts of ICT and of device renewal in particular, go far beyond carbon emissions. While computer science research has long focused on quantitative approaches to mostly direct emissions, there is a growing recognition of the rebound effects together with indirect quantitative and qualitative effects of digital technology.

The mining processes for the extraction of digital minerals, followed by their purification for chip and electronic device manufacturing, consume great amounts of fossil energy, water and chemicals polluting the habitats, degrading the biodiversity and the life of surrounding populations, on top of causing political, economical and social crisis [29, 30]. For example the cobalt and coltan extraction in the Democratic Republic of Congo—also called *conflict minerals*, used in smartphones, batteries(cobalt) and capacitors in all electronic devices(coltan)—has played an important role in the political tensions and conflicts in the whole Central African region, as Vogel in his book *Congo's Conflict Minerals: War, Profit & White Saviourism*, based on in-situ analysis, shows [31]. He describes how the problems were deepened by programs pretending to solve them, such as *Clean Sourcing* or *Conflict-free ethical mining* certifications that *"prioritized satisfying the conscience of Western consumers rather than bettering Congolese livelihoods [...] permitting some Fortune 500 companies to secure extractable mineral reservoirs at the expense of expendable bodies*".

Another example is that of the mining of Lithium—used in most of the modern batteries in digital devices but also electrical transport engines—which exacerbates water scarcity and health issues in Nigeria [32]; indigenous human rights in Chile [2] and raises socio-environmental and democracy governance issues in France [3]. Similarly, the purification of minerals and chip production causes

[2] Resisting in the age of lithium: the right to a healthy environment in Indigenous territories in Chile, Feb. 2026, UN Office of the High Commissioner, archived.

[3] See the Lithium issues of the online journal *Les Temps qui restent*, 2025 - 2026, archived, last accessed in April,

water pollution and scarcity issues in Grenoble France, home to STMicroelectronics and Soitec semiconductor manufacturing plants [4], while new chip factories in the United States significantly affect land, energy and water supplies as well as airways [33].

The above mentioned issues are only some of the socio-environmental issues of digital device or infrastructure production. In the article *Reflections on the impact of digital infrastructure and racism in traditional communities*, author Neves-Barros talks about the *"environmental racism"* of digital infrastructure and e-waste management in the Global South, about its *"predatory business model [...] that recreate new forms of slavery and undermine autonomy"* [34]. The *Digital colonialism* studies emphasize the instrumental role of digital rights activists in knowledge production [35], while new joint initiatives by scholars, activists and policy makers at the intersection of the climate crisis and digital rights emerge [36], contributing to a broad and situated reflection on socio-environmental, systemic and political issues at stake.

1.1.2 Device renewal: end-of-life and e-waste

As for the end-of-life of devices, it is important to note that the electronic waste (e-waste) problem remains today largely unresolved and insufficiently assessed [37]. In their dedicated program called *The Global E-waste Monitor*, the United Nations define e-waste as *"any discarded product with a plug or battery"*. They add that e-waste *"is a health and environmental hazard, containing toxic additives or hazardous substances such as mercury, which can damage the human brain and nervous system"*[5]. The latest report from the program in 2024 showed that the amount of e-waste that we produce worldwide is constantly increasing (21% of increase in the 5 years from 2015 to 2020). Data from 2022 in the report show that e-waste is rising five times faster than the amount being recycled, and that only 25% of it is being officially recycled worldwide.

In his book called *Reassembling Rubbish: Worlding Electronic Waste* [38], Lepawsky questions the official and legal definition of e-waste as a post-consumer problem, and describes how unconsidered waste occurs at all stages of electronic devices existence. Even within the official post-consumer framing, e-waste, is often left out of footprint considerations. Authors Braungart and McDonough [39] as well as Mewes [10] argue that reasons for that may be the fact that after products are sold to consumers, they are no longer part of major considerations or of externality evaluation—the problem is left to municipalities, consumers, or whoever can make value off the waste.

I experienced how the economic value of the waste was indeed a decisive waste management factor, when, in November 2022, together with my research team, we visited Loxy, a waste electric and electronic equipment (WEEE) treatment center in the Île-de-France region[6]. There were two types of e-waste at Loxy, the items that could be resold, and the rest—considered 'obsolete' on the waste market—that ended up being shredded or landfilled. For example for CPUs (Central Proccessing Units)—shown in Figure 1.2—decisions on whether they were to be resold or shredded came in the form of regularly updated lists of models that were in demand on the resale market. There were no technical considerations, such as performance for example: more powerful processors were marked for shredding, maybe because they were older, while less powerful but newer ones, were

2026.

[4] Grenoble demands 'water not microchips!', Le Monde Diplomatique, July 2023, last accessed in April 2026.

[5] The UN Global E-Waste Monitor 2024 report, last accessed in April 2026.

[6] Newsletter n°6 of the Limites Numériques team: End of life & digital waste, March 14, 2023, last accessed in April, 2026.

sometimes marked for resale. Part of the employees at Loxy were dedicated to monitor and analyse the resale market and to use eBay as a resale online shop.

Figure 1.2: CPU waste bin at Loxy WEEE treatment center.

The above non-exhaustive overview of the impacts associated with device production and renewal—from their manufacturing to their end of life—leaves little room for doubt: we must take action to address this problem. Extending the lifespan of devices appears to be the minimum remediation strategy—certainly insufficient—but let us start from there.

1.2 Research questions: What drives software obsolescence and how to remediate it?

Recent studies define the lifespan of devices in terms of obsolescence [40, 41], viewing it not only as a technical or engineering quality, but also as being influenced by socio-cultural and socio-economic factors. In this framing of a socio-technical ecosystem, obsolescence is not simply about technology existing within society, but rather a dynamic interplay where social structures, human behaviors, and technological artifacts each shape and influences the other.

Environmental sustainability issues also arise in software engineering and its maintenance. Software accounts for unsustainability [42] and affects that of IT products and services they deliver[43, 44]. As software is increasingly embedded in almost every digital device or device elements—personal computers, smartphones, tablets, connected devices such as cars, watches, home appliances, but also servers in data centers—understanding software obsolescence has become crucial in understanding

obsolescence and addressing sustainability issues of devices.

Device renewal is particularly important in the case of smartphones, that appear to be the poster child of 'disposable technology', i.e., devices that are neither maintained, nor meant to be repairable or recyclable, but rather replaced [45, 46, 47] and causing e-waste [48]. Magnier and Mugge's survey of European consumers shows that decrease in performances and software problems are the main factors leading to a perceived loss of value in devices [46]. In our article called *Obsolescence Paths: living with aging devices*, we show how software issues play an important factor in smartphone obsolescence for users: upgrades, storage issues, and malfunctions that accumulate, showing precisely that hardware and software obsolescence are entangled, and should be considered together.

Smartphone manufacturers offer and promote new models of devices at increasingly frequent rates. If we take a look at the two world-leading smartphone manufacturers, Apple and Samsung, we see that Apple released a new model every year since 2007, and since 2017 three to five models a year [7]; while the latter released an average of 9 new models per year from 2014 until 2018, and not less than twenty new models per year since 2019 [8]. In parallel, new versions of Operating Systems (OS) for smartphones are also released frequently. The OS market is largely dominated by Apple iOS (29% of worldwide OS mobiles) and Google Android OS (70%) [9] since 2012, after the collapse of the Nokia SymbianOs, collapse that began soon after the first iPhone was commercialized in 2007, followed by the first Google Android phone commercialized in 2008. Since then, both iOS and Android OS release a new major version every year [10].

In order to investigate what drives software obsolescence, our first case study presented in chapter 2, focuses on the Android ecosystem, the world's most widely used operating system, where devices are rarely updated more than two years after their release. We investigate what hinders Android development and maintenance issues fostering obsolescence.

As a second case study, in order to better understand software maintenance and remediation strategies to software obsolescence, we chose to investigate the Debian OS, a widely used, more than 30 years old, operating system also based on the Linux kernel, maintained by a community organized as a non-profit organization. Debian has a reputation for durability, stability, and reliable maintenance among users, developers and professionals worldwide. We present this study in chapter 3. In the same time we analyze how, driven by a concern for longevity, some alternative free open-source mobile actors are implementing remediation strategies, in particular software development strategies, to offer longer support to smartphone devices and extend their lifetime in both chapters 2 and 3.

Last but not least, in this PhD thesis we drew from previous and ongoing work on obsolescence from science and technology studies, in order to better understand how obsolescence manifests itself and plays out in our society, from a wider socio-technical and socio-economic perspective, that we present in chapter 4. This chapter is a collaboration with obsolescence scholar Jeanne Guien [49].

By doing so, we seek to answer to the following questions:

- When analysing the Android software ecosystem, its development and maintenance practices, what can we learn on software related issues causing obsolescence? (chapter 2).

[7] Timeline of iPhone models, Wikipedia, last accessed in Jan, 2026.
[8] List of Samsung Android smartphones, Wikipedia, last accessed in March 2026
[9] Usage share of Operating Systems Worldwide, Wikipedia, last accessed in April, 2026.
[10] Android version history and iOS version history, Wikipedia, last accessed in April, 2026.

- How is maintenance organized in a community driven, free and open source, socio-technical system such as Debian? What fosters and what inhibits maintenance? (chapter 3).
- What can we learn from mobile alternative systems and Debian regarding software long-term maintenance, obsolescence remediation strategies and device life extension? (chapters 2, 3 and 5).
- In the overall socio-technical landscape, what drives digital obsolescence and how do software and hardware drive the obsolescence of each other? (chapter 4).
- What obsolescence remediation strategies and recommendations do we draw from this study? (chapter 5).

This thesis is organized in chapters, as follows:

1. This introduction, presenting the research context, research methodology and background on obsolescence (chapter 1).
2. Chapter 2, called *Producing software obsolescence: the case of Android OS* maps the Android ecosystem, its actors and the plays between them in producing software. We show how and where maintenance occurs in the Android ecosystem, yielding to mobile device obsolescence. In parallel, this chapter shows maintenance strategies of some mobile alternative actors from FOSS systems and phone companies.
3. Chapter 3, called *Maintenance strategies in Debian* shows our in-field study among the Debian OS community, analysing how maintenance is structured, what sustains long-term maintenance in Debian, and what puts it at risk. This analysis completes and expands that of alternative mobile strategies and gives a better comprehension of software development and maintenance practices to avoid obsolescence.
4. In chapter 4 we expand our discussion on obsolescence in order to understand how hardware and software drive the obsolescence of each other, and how obsolescence serves as a model for the digital capitalist economy.
5. Last but not least, in chapter 5 we summarize the findings of our two case-studies: the Android ecosystem and Debian OS. Building on our broader analysis of obsolescence in chapter 4, we discuss briefly what we learned as factors fueling obsolescence in and by software on one hand, and what maintenance strategies towards longevity we analysed as obsolescence remediation strategies. To conclude, we formulate a number of recommendations on coding practices on one hand, and on regulatory measures needed to enforce sustainable coding practices and avoid software obsolescence.

1.3 Background on obsolescence

Obsolescence often refers to the state of a material, device, part, software, product or even infrastructure that is no longer maintained or in use, even if it is still functional. Obsolescence has been the subject of various research studies, and different types of obsolescence have been defined. When I first began this thesis, my first naive attempt was to find a definition of obsolescence. During my

work I found many of them, and realised that obsolescence is, to say it in the words of Proske & Jaeger-Erben, *"not a neutral description of a particular state of an object"* [50].

Up to this moment, I continue to find new definitions of types of obsolescence, each differing from the context in which they were used and analysed, the institution or entity from which they come, their position in the broad technical, political, economic and social spectrum and their own interests in the subject. I will not give another definition of obsolescence, neither will I succeed in showing the multiple definitions I found. Instead, I propose to discuss here the approaches that have shaped my understanding of the subject, highlighting those that have sparked my critical thinking or that have further inspired and motivated this study.

1.3.1 Managing software obsolescence for industry needs

Software maintenance has been a long-standing concern in software engineering research and industry. While early methodologies attempting to deal with Commercial Off The Shelf (COTS) obsolescence are related to hardware and spare-parts, software obsolescence became a topic of concern in the early 2000s [51, 52, 53]. One of the most widespread software obsolescence definitions in computer science comes from Peter Sandborn, a software engineering researcher, whose work in obsolescence is centered on methodologies for predicting, managing and avoiding it in an industrial COTS setting. He defines it in the following terms [54]:

> "*Software obsolescence (more specifically COTS – Commercial Off The Shelf software obsolescence) is generally due to one of three main causes:*
>
> 1. *Functional Obsolescence: Hardware, requirements, or other software changes to the system obsolete the functionality of the software (includes hardware obsolescence precipitated software obsolescence; and software that obsoletes software).*
> 2. *Technological Obsolescence: The sales and/or support for COTS software terminates:*
>   - *The original supplier no longer sells the software as new (end-of-sale)*
>   - *The inability to expand or renew licensing agreements (legally unprocurable)*
>   - *Software maintenance terminates - the original supplier and/or third parties no longer support the software (end-of-support)*
> 3. *Logistical Obsolescence: Digital media obsolescence, formatting, or degradation limits or terminates access to software."*

To summarize, this definition states that software obsolescence in digital devices can be the result of an update of one software rendering another obsolete, or the termination of technical, licensing or contractual support of the software, or the update of a software that cannot be executed in a certain hardware.

In all of his scientific papers or books, Sandborn likes to repeat the following phrase, attributed to Bill Gates—founder and CEO of Microsoft— putting it usually in the first paragraphs of the introduction, as if to justify the relevance and urgency of software obsolescence management: *"The only large companies that will succeed are those that make their own products obsolete"*. I wanted to read more about obsolescence from Bill Gates so I naturally looked for the context in which he had said or written this phrase. It was quite a long journey to find the original statement though,

maybe because it was said a long time ago, but what struck me in the process of finding the original source, was that this quotation was repeated a lot. I found it on all kind of media: research articles in different fields, books and essays, financial journal articles, well-known magazines like the Times, and even in a catalog selling hunting bows (shown in Figure 1.3).

Figure 1.3: Screenshot of a hunting bows selling catalog from 2003 containing the quote on obsolescence attributed to Gates. Source: Archive of 2003 Mathews catalog, p.2, archived in Dec. 2025.

In his papers, Sandborn references the quote as taken from a news article from 2003, called *The Bill Gates Method*, in what appeared to be a magazine of Advancement Placement Team, a company specialized in accounting and financial placement counseling (see Figure 1.4). The source article listed 8 key-points of an alleged Gate's method in handling business, had no authorship, and although the quote from Gates appears in a longer form, it is not referenced.

After hours and days of search, in journal archives, websites and their archives, books, articles, I could not find the original context containing this exact quote from Bill Gates. Maybe this popular quote had been altered while being used in so many contexts. But in an autobiography book, co-authored with C. Hemingway in 1999, called *Business @ the Speed of Thought: Using a Digital Nervous System* [55, p. 152] Gates talks about obsolescence at Microsoft in a quite similar way, aligned with the referenced quote:

> "*In three years, all the products my company makes will be obsolete. What matters is whether we make them obsolete, or whether someone else will do it for us.*"

If obsolescence seems to be, by Gates words, what drives production and market competition at Microsoft and more largely in software companies, the fascination that this vision holds in other fields—as I discovered while searching for the original quote—is just as interesting. It demonstrates that obsolescence is widely accepted as a successful business strategy, as a way to handle business, and that mastering it is a winning strategy in business to business competition. For Sandborn, this is a way of convincing of the importance of obsolescence management in industry. Obsolescence is treated by both Gates and Sandborn as something frightening, a fatality that threatens business models, but that in the same time fosters them, if one knows how to manage it and use it to his advantage.

In his research work, Sandborn develops models and methodologies to predict software obsolescence through forensics and prevention methods, that he applies to several industrial contexts in order to avoid it, working with them and for them. Other scholars have built on his models of

The Wayback Machine - https://web.archive.org/web/20031128233537/http://www.placementpartner.com:80/apt%20news/news7-21-03.html

APT News — July 21, 2003

"Opportunity is missed by most people because it is dressed in overalls and looks like work." ~ Thomas A. Edison

Available Applicants

ACCOUNTING ASSISTANT ~ SEEKS $12/HR
Fast, reliable candidate has over three years' accounting experience handling a variety of duties at the corporate office of a small furniture store chain. Responsible for company's bookkeeping including A/P, A/R, keying all invoices and payments from customers, new vendor/item setup, collections, bank recs, and sweeps (transferring money from each store to main account). Utilized QuickBooks and Excel daily. Pleasant with a can-do attitude. Ready to get started today!

CREDIT/COLLECTIONS ~ SEEKS $14/HR
Using superior customer service skills, this credit analyst confidently handles escalating calls to resolution. Has over five years' experience with large commercial collections. Makes decisions regarding account increases, reductions, holds, and collections working within guidelines. Keeps tax certificates and files intent to lien letters. Has also worked in shipping departments and understands the full cycle, from delivery to billing. Resolves accounts using knowledge of POD's, BOL's, and carrier claims. Experienced with AS400, Oracle, Excel, and Access.

JUNIOR ACCOUNTANT ~ SEEKS $25K
BS Accounting, 3.4 GPA. Friendly recent grad is ready to put degree and experience to use! Gained valuable skills working part-time through college as an A/R clerk at a publishing company and through a tax internship. Maintained/adjusted customer accounts after payments/credits, posted payments, collections, account recs, and prepared federal/state income tax returns. Eager to get a foot in the door. Available immediately!

BOOKKEEPER/OFFICE MANAGER ~ SEEKS $35K
With ten years' experience as business manager for an interior design studio, this polished, articulate candidate is ready for a new challenge! Handled A/P, A/R, sales tax, invoicing, G/L entries, payroll and 940 deposits, bank recs, P&L reports, inventory control, and staff supervision. Worked closely with outside CPA to produce accurate monthly financial statements. Versed in MYOB and QuickBooks. Super flexibility and initiative.

CONTROLLER ~ SEEKS $55K
CPA, BS Accounting. Sharp go-getter has experience setting up and managing entire accounting functions for a start-up manufacturer as well as five years' experience as controller for mid-size distributor. Responsible for preparing monthly financial statements, bank relations and reporting, tax filings, purchasing, HR functions, lease documentation/discounting, business plan development, and overseeing tax/financial audits. Versed in SBT and Peachtree. Great at managing processes and people. Available now!

The BILL GATES Method

1. **THINK AND DO**
   Bill Gates says it is fun to learn new things. He spends time doing 'think weeks,' where he reads all the stuff smart people have sent him. But to Gates, education is a means to an end - and that end is action. Gates says: "Let's use our heads and think - and do better software than anyone else."

2. **ONLY HIRE VERY, VERY SMART PEOPLE**
   Gates always insisted on hiring only the very brightest talent, or what Microsoft calls, "high IQ people." He believes the biggest problem with hiring mediocre people is that they take up space that could be occupied by brilliant people.

3. **KNOW YOUR BUSINESS BETTER THAN ANYONE ELSE**
   Bill Gates' personal technological knowledge is one of the key components of Microsoft's success.

4. **CREATE THE INDUSTRY STANDARD**
   Microsoft's company slogan is, "We set the standard." Gates has been known to buy out whole companies in order to gain their technology and brain trust. He has been insistent that his company be both first and best in the marketplace.

5. **GET TO THE POINT**
   Bill Gates hates to waste time. He is not afraid to say some ideas are dumb and not worthy of his time.

6. **WORK LONG, WORK HARD, WORK SMART**
   Although Bill gates could easily retire quite wealthy, he prefers to work 12 to 16 hour days. Microsoft demands that deadlines are met, performance is high, and tangible results achieved.

7. **WHEN YOU CAN'T ORIGINATE, ELEVATE THEN SELL, SELL, SELL**
   Bill Gates has bought a lot of small companies with some good ideas. He has said since his youth, "Let's call the real world and try to sell something to it."

8. **FUTURE THINK**
   Gates believes in order to beat competition you have to be several moves ahead. Gates states, "If you sit still, the value of what you have drops to zero pretty quickly. Every company is going to have to avoid business as usual. The only big companies that succeed will be those that obsolete their own products before somebody else does."

Figure 1.4: The Bill Gates Method, 2003, Internet Archive of APT News, last accessed in Dec. 2025.

managing obsolescence in numerous industries such as the military, aviation, rail, automotive or industrial machinery manufacturing, all concerned with ensuring the longevity of the software and hardware systems they integrate in their organizations and work processes [56, 57, 58].

The management of obsolescence software issues seems to be taken very seriously by companies too, eager to manage their inner chain of production or operations involving software. During this thesis I found out about the International Institute for Obsolescence Management (IIOM), a not-for-profit organisation that *"exists to advance the science and practice of Obsolescence Management"*[11]. The institute has chapters in many countries, the French Institute of Obsolescence (IFO), the UK and Ireland one, in the United States, etc. All institutes are dedicated to discussing the best methods for 'managing' obsolescence, building international standards and links between companies. The French chapter's annual *Obso-Days* is a multiple days conference bringing together professionals from the aerospace and defense industries, as well as, quoting, *"obsolescence experts"* convinced that

[11] International Institute for Obsolescence Management, last accessed in March, 2026.

managing obsolescence is a *"vector of agility and, alongside innovation, a lever of sustainability"*[12]. They even participate in a specialized master's program, within the Paris Institute of Mechanical Engineering, titled *Sustainability-Obsolescence-Scarcity*, enabling students to become *"obsolescence experts"*. During this study we attended some of the online seminars from the UK and French chapters of IIOM, and followed for several months online news and activity reports. We noticed that these institutes and conferences are organized and attended mostly by companies, their managers and executives, and that all actors tackle inner operational and production obsolescence problems for their companies, trying to avoid it and manage it better. We did not encounter obsolescence management for consumer goods, digital devices or software being sold by these companies in these meetings, this was clearly not a goal. It was almost always about the companies inner production and operational supply chains, or industrial devices where durability and reliability are highly involved—as in trains, airplanes or military equipment.

We further discuss this apparent antagonism in chapter 4, and analyse how it happens to be the two facets of the same: managing obsolescence is about knowing how to avoid it in some contexts (such as industrial needs), or exacerbate and provoke it in some other ones (such as consumer goods production and sale).

1.3.2 Software obsolescence from a systemic socio-economic perspective

By studying the market and social history of some everyday objects, such as disposable cups, paper tissues, advertisement store fronts, deodorants, disposable menstrual products and also smartphones, Guien [59, 60] argues that obsolescence is at the very heart of the business model for consumer goods and has been theorized as such since early management studies in the 1930's. From this historical and socio-economic perspective presented by Guien, basing her analysis—among other work— on Susan Strasser's *Waste and Want: a Social History of Trash* [61] firt published in 1948, we understand that obsolescence should not be understood as a state of being for objects or products, but rather as being produced, managed and planned, in a society structured around the constant ongoing production of disposable products and the promotion of consumption of new ones. Guien's work was an important inspiration in understanding the socio-economic and socio-technical aspects of obsolescence for this PhD thesis, in particular her 2021 book *Consumerism through its objects*[13], based on her PhD thesis in philosophy of science [59].

I had the chance to work with Guien during this PhD in writing a book chapter on obsolescence as a business model of digital capitalism [49]. Chapter 4 is an extended version of this book chapter, where I add further personal observations and material.

1.3.3 Planned obsolescence definitions shaping ambivalent narratives

The idea behind what has been called *planned obsolescence*, is one of suspicion towards companies in deliberately shortening of the lifespan of a product to force people to purchase replacements. Proske et al., define it as the *"deliberate shortening of the product lifetime by the manufacturer"* [62]. The term is believed to come at least as early as 1932 with Bernard London's pamphlet called *Ending the Depression Through Planned Obsolescence* [14], but examples of use under different names have been

[12] See IFO's Obso Days 2025, Institut-obsolescence.info, last accessed in Dec. 2025.
[13] Titre traduit du français *Le consumérisme à travers ses objets.*
[14] Bernard London's pamphlet, 1932, Wikimedia Commons, last accessed in April 2026.

analysed in the automobile and cycling industry as early as the end of the 19th century, beginning of the 20th century [63]. In his pamphlet London wanted the US government to impose a legal obsolescence on personal-use items, to stimulate purchasing after the 1929 economic crisis.

In 1986, economy scholar Bulow, provided details on the way planned obsolescence occurs and is linked to monopolies and oligopolies by showing that: on one hand monopolist companies will prefer to plan obsolescence by selling *"goods with inefficiently short lives*" and *"to rent rather than sell*" the durable ones; while the oligopolist companies could opt for selling more durable goods if they are entering the market, but otherwise will have a general incentive *"to collude to reduce durability*" of the goods that they sell [64]. The role of monopolies and oligopolies in increasing planned obsolescence was confirmed afterwards, for example in Orbach's work [65].

In 1954 Brooks Stevens, an American industrial designer popularised the term in a talk at an advertising conference in Minneapolis, but his definition of planned obsolescence as an industrial designer's mission was novel, as it highlighted the act of instilling the desire for replacing goods into the customers, without mentioning the other systemic economic production aspects: *"Instilling in the buyer the desire to own something a little newer, a little better, a little sooner than is necessary."* [15].

In 1960, the journalist Vance Packard published a consumerism critical book, called *The Waste Makers*, describing *"the systematic attempt of business to make us wasteful, debt-ridden, permanently discontented individuals*" where he divided the definition of planned obsolescence into two subcategories: planned obsolescence of functionality, which corresponds to the historical definition, and a new one, that he calls *"the planned obsolescence of desirability*" [66, chapter 7]. He defines it as:

> "*a strategy [...] to persuade the public that style is an important element in the desirability of one's product. Once that premise is accepted, you can create obsolescence-in-the-mind merely by shifting to another style.*"

Also called *psychological obsolescence* or *obsolescence of style* by others, this definition builds upon the one from Stevens, using design and advertisement, but Packard takes the opposite view here, vehemently criticizing in his book the psychological manipulation of consumers and going as far as pointing out the resource exhaustion and impacts on waste of this planned obsolescence.

What struck me most while reading Packard, is that he notes how this obsolescence of desirability and style, masks and hinders genuine product improvement, by making designers focus on desirability and style factors, or masking defaults with cosmetic changes, rather than focusing on improving functionality. This critical approach of obsolescence as a brake to real improvement seems at first, antagonist to the 'myth of innovation'. Innovation is often presented as an improvement in quality of functionality or as technological progress. In the same time, as Gates does in his above quotation, some accept and welcome obsolescence in order for innovation to take place, saying that obsolescence fosters market competition and technological progress [67]. The term innovation in itself contains the idea of novelty, a novel product replacing older ones in quick innovation cycles, each following s-curves, the standard model of measuring technological product adoption and spread. The more obsolescence we have, the vaster the replacement of products, and the better and quicker innovation spreads. Hence, in this narrative, obsolescence helps innovation.

But what Packard criticizes here, is that this innovation is only an illusion, as style or desirability obsolescence masks and prevents some real product improvements, which should be one of the main

[15] The Milwaukee Art Musem (MAM) archives on Brooks Stevens, mam.org, last accessed in Feb. 2026.

characteristics of technological innovation. Innovation and obsolescence become different aspects of the same problem: the pursuit of economic profit at the expense of technical quality. We find this antagonism also in the *Creative Destruction* concept that economist Schumpeter coins: the idea that new methods of production survive by eliminating existing ones, creating the conditions to growth in capitalism and the free-market—better products, greater profits—but rely on destruction and obsolescence to achieve it [68]. Innovation is not about product quality or improvement: it is a business model relying on obsolescence. This link between obsolescence and innovation, is also what we have tried to describe with Guien in [49] and chapter 4.

In an article called *Toward a Throw-Away Culture: Consumerism, 'Style Obsolescence' and Cultural Theory in the 1950s and 1960s*, Whiteley traces back how the perception of this psychological/style/desirability planned obsolescence has changed through the years [69]. In the 50's it had a positive cultural perception in the US. Whiteley describes then *style obsolescence*, as the *"[...]'American way' of design—high consumption, rapid obsolescence and design as a social language*". Continuing into the 60's, he describes how criticism of consumerism and obsolescence began to rise, pointing out the resource exhaustion, waste management and its social and environmental negative impacts.

It is interesting to note here that in France, the definition given in 2020 to obsolescence from ADEME (Agency for the Environment and Energy Management), a French state institution in charge of the ecological transition, in the report called *Economic Assessment of Extending the Lifespan of Consumer Goods and Capital Goods*, also used two aspects in describing obsolescence. The first aspect is the breakdown. The second one is what ADEME called 'cultural obsolescence' or 'perceived obsolescence' [70, p. 10]. Let us cite the translation of the definition of the latter:

> "*Cultural or perceived obsolescence involves discarding equipment even though it still works, in order to keep up with changing needs, trends, or innovations. It can also lead to hoarding, a phenomenon particularly evident with textile products. It is accelerated by constantly evolving marketing strategies and ubiquitous advertising.*"

Not only does this definition strip away the critical systemic economic aspects of planned obsolescence examined so far, but it makes use of a 'culture' without defining it, as if culture was the same among countries, different populations or even socio-economic categories of a same region. Culture here appears to be defined by a mix of fashion, style, and innovation, where the user-consumer is the main actor having leverage on the obsolescence, seen as an individual consumer problem. This report is followed by numerous videos and articles produced by ADEME containing a slew of guilt-inducing advice urging users to become *"good"* or *"responsible consumers*" by repairing items subject to breakdown or by resisting the urge to replace them when 'perceiving' obsolescence[16].

This definition was revised by the ADEME in a new report from 2026, aimed at researchers, called *Consequential analysis of eco-design levers for digital services* [71, p. 51]. The definition of obsolescence is here a new one, as follows:

> "*Generally speaking, a product's "obsolescence"—that is, its loss of value before it wears out physically—encompasses all the reasons for discarding it.*
>
> - *'Functional obsolescence' refers to a product no longer meeting new expected uses, for technical, regulatory, or economic reasons. This is the case, for example, when*

[16] See for example this article or these videos on cultural obsolescence by ADEME, last accessed in April 2026.

a product suffers an irreparable breakdown.

- *'Evolutionary obsolescence' refers to the fact that a product no longer meets the desires of users who wish to purchase a new model due to changes in functionality or design. Today, the average individual usage period for smartphones in France is estimated to be between 23 and 37 months, whereas the potential lifespan is 5 or even 10 years for certain mobile devices."*

This time, the definition taps into user *"desires"* (*"envies"* in French), and their *"wishes"* (*"souhaits"* in French) to replace. Moreover, this obsolescence is now described by the concept of 'evolution', not defined, but presented a few words later as regarding *"changes in functionality or design"*. These changes are not described, their causes or socio-economic contexts not analysed from the perspective of markets, industries or business factors, but always from the perspective of users, their desires and wishes, vague undefined emotions that contrast with the rationality suggested by technological evolution. Even the definition of the functional obsolescence, the usual historical perspective of obsolescence presented above, has shifted to the users' point of view, by saying that obsolescence happens when their *"expected uses"* (*"usages attendus"* in French) are not being met. This vague and undefined expression refrains from challenging the industry ways of producing goods, but—once again— holds the users as responsible for obsolescence.

The recommendations of the ADEME report in combating 'evolutionary obsolescence' are also focused on user-consumer responsible behaviour, encouraging them to somehow limit their own irrational influence from fashion and trends (but only *"to some extent"*): *"by consuming responsibly, in line with one's actual needs, by maintaining one's products, and by limiting, to some extent, the influence of trends that encourage premature replacements"*.

It is precisely the question of 'desire' that obsolescence scholar Guien analyses in her book *The Desire for Novelty: Obsolescence at the Heart of Capitalism* [72][17]. Far from denying the existence of this 'desire' for new products and novelty, Guien traces its historical origins from the colonization and the importation of products considered exotic—during which fashion truly began to develop in France—to the evolution of this phenomenon up to the 18th century with the advent of industrialization and its roots in the capitalist system. Guien shows here how novelty, far from being a simple aesthetic or social phenomenon, serves an economic logic: that of planned obsolescence. She then continues into examining how marketing, design, and advertising in the 20th century transformed novelty into a commercial imperative and how the *"promise of novelty"* has fueled a consumer society that, under the guise of modernity and comfort, hides harmful environmental and social effects. The way consumer desires and behaviour affects planned obsolescence have also been analysed by Proske et Jaeger-Erben in their analysis of modular smartphones for longevity [50]. Here the authors define obsolescence as *"a process of communicatively and materially devaluing a product"*, where *"the human-object relationships are not only key to understanding obsolescence, but also essential for strategies to prolong product lifetimes, to increase their functionality and, through this, the value of a product to its user"*.

Last but not least, as Barros et al. observe how, in recent years, there is a change in the patterns of planned obsolescence strategies employed by digital companies, shifting from aesthetic to technological obsolescence [73]. By taking the example of the mobile phone industry, authors highlight how new planned obsolescence strategies are embedded in product characteristics and

[17] Traduction du titre original en français *Le désir de nouveauté: l'obsolescence au coeur du capitalisme.*

functionalities, much more than on style. In chapter 4 we argue that style, product characteristics or functionalities, are all opportunities for obsolescence in the digital capitalism economy, invoked sometimes in turn, sometimes together, driving the obsolescence of each other. The slimness of the latest iPhone and Samsung S25 phones in 2025 was once again a physical characteristic presented as a style, advertised as being the latest 'trend'. Even though this same slimness has caused phone bending issues and scandals in the past, such as the iPhone 6 and 7 *bendgate* that have been the subject of consumer legal complaints [18].

1.3.4 The misleading 'programmed obsolescence' regulation and narrative

In French, the term "programmed obsolescence," mistranslated from the English 'planned obsolescence' reinforces the notion of a secretly programmed hidden flaw causing obsolescence in electronic devices. It suggests that the model of obsolescence is a software-programmed malfunction inserted in devices. Thus, it is often presumed in France that obsolescence is something programmed by software, hidden in the machine, that users, journalists or technicians have to investigate in order to discover.

This tendency is also present in the French law. In 2015 France was the first country to make planned obsolescence a criminal offense by law n° 2021-1485. Amended in 2021 this law simplifies the definition, but also emphasises the 'programming' aspect by defining it as *"the use of techniques, including software, by which the party responsible for placing a product on the market aims to deliberately reduce its lifespan*". Complaints filed in France have targeted smartphone brands such as Apple, accused of intentionally slowing down, deteriorating the performance of devices by performing a software update. The problem with this definition is that it relies on 'intentions'. Not only it is difficult to prove intent in court, but to add to the burden, the burden of proof lies with the victims. That is how, in 2025, 10 years after being put in place—despite several complaints filed against Epson, Apple or HP—no court ruling has yet been issued. This fosters a sense of impunity and raises concerns about democratic trust among civil society organisations[19].

The problem with this definition is not only a regulatory one. As we argue in our work with Guien [49], and as stated in chapter 4 this definition ironically hides the unhidden aspects of obsolescence, that companies, advertisers and digital companies have built, theorized, proudly presented as marketing or economic models, using different intertwined opportunities to create obsolescence.

Last but not least, let us here mention the *unplanned obsolescence* and consequences on digital technology and infrastructure in the case of natural disasters on the rise amid the environmental crisis. Authors Di Natale et al. argue for reuse of chips at design time as a sustainability key solution to shortage [74] while Courtillat-Piazza, Quinton & Marquet reflect on the un-sustainability of growing digitization trends in case of long-term chip shortage [75]. Authors Jang et al. [76] argue that in a collapse scenario, software presents challenges harder to mitigate, as the detrimental effects of long term disconnection, software data corruption, and malware are numerous and have little to not at all been analysed and addressed. I believe that here again, FOSS systems have a significant role to play. Their unofficial status, development behind the scenes, have helped FOSS

[18] Apple Confirms It Knew About iPhone 'Bendgate', 24 May 2018, last accessed in July 2025.

[19] *Programmed obsolescence: Ten years after the law was passed, no court ruling has yet been issued*, op-ed, August 2025, Le Monde, last accessed in March 2026.

systems in being more resilient, adapting to crises and relying less on centralized big infrastructure or dominant technology such as Google services. This is for example the case of UnifiedPush, an open standardized protocol and set of tools that can replace the Google Push Services in connected devices, that we further describe in chapter 2. The collaborative, open source, often decentralized nature of FOSS alternatives facilitates modifications and adjustments as needed. It is also by making use of decentralized public archiving solutions such as Software Heritage[20] that can help ensure software resilience, in situations of crisis where software binaries that run on centralized platforms are not available anymore.

1.3.5 Software maintenance, digital commons and alternative solutions for sustainability using FOSS

In the software industry at large, despite the emphasis on innovation in public discourse, much of the software work is centered on maintenance. As Webster et al. point out, *"maintenance is an unavoidable activity required to keep systems synchronized with the reality they are modeling, a reality that changes continuously*" [77]. As the environment in which the code evolves, the functioning code "decays" [78]. Synthesizing surveys in the literature, Canfora and Cimitile estimate that software maintenance consumes between 60% and 80% of the total life cycle cost of software projects [79]. But they also note that, according to these same surveys, a large share of these maintenance costs (75% to 80%) relates to enhancements rather than corrections. In this sense, maintenance goes beyond correcting bugs, with a large portion related to evolutive maintenance.

Wieser et Tröger have shown that for long-lasting products, continual software updates and support become very important factors, and that discontinued software support is a replacement reason [80]. But on the other hand, from the user's perspective, little is done to prepare them for the update and upgrade: these processes appear as difficult to manage to users, with little to no information or user interface designs provided for better understanding or control [81]. In FOSS systems, maintenance is central to the work of the communities that develop them. In their work on the labor of maintaining and scaling FOSS systems, Geiger, Howard and Irani detail how maintainers in the community have a central role, and how *"maintenance is not only about repairing and fixing. It is crucially about updating and changing to stay relevant*" [82].

Alternative solutions in divesting from Big Tech companies, are also considered by scholars as ways of avoiding or stopping their unsustainable social and environmental impacts [83]. In *The Commons: A Social Form that Allows for Degrowth and Sustainability*, Euler points out how the commons, social forms *"constituted by social practices (commoning) that are based on voluntariness, autonomy and needs-satisfaction*", do not have an inbuilt growth compulsion as is usually the case in capitalism, and thus better allow for sustainability. In their paper *Digital Commons for the Ecological Transition: Ethics, Praxis and Policies* authors Shulz et al. discuss precisely how some well known FOSS systems such as Linux, Wikipedia or OpenStreetMap, that have established themselves as digital commons, deal with environmental concerns and ecological transition. The authors identify patterns of sustainability in digital commons, challenges and limitations and identify ways forward in achieving long-term environmental sustainability.

[20] The Software Heritage initiative website, last accessed in April 2026.

1.3.6 Social practices around maintenance and care

Beyond the software industry, scholarship on “maintenance and repair studies" developed at the intersection of various fields, from geography [84], to sociology [85]. This work broadly attempts to tie the literature on Care to the material consideration related to maintenance and repair [86]. It draws notably on Anne Marie Mol [87] and Maria Puig de la Bellacasa work on articulating *“maintenance and repair as processes dedicated to restoring order*” [86]. This approach invites to consider tools and technical artifacts not as solid and permanent but as fragile objects, with a constant need for care to remain functional. This scholarship has studied infrastructures, cars, homes, and more, but is much more limited when it comes to software and ICT. Let us note here Serra-Fontalvo’s review of literature and practices addressing obsolescence in product design [88]. In software, Marisa Leavitt Cohn’s study of maintenance at NASA [89] is an exception.

Daniela Rosner and Morgan G. Ames, building upon field studies of repair practices, also note that *“breakdown and repair are not processes that designers can effectively script ahead of time; instead, they emerge in everyday practice*” [90]. This relates to the fragile nature of digital technologies emphasized by Steven Jackson: breakdowns are constitutive of technological systems and happen all the time [91]. But their identification and the worthiness of maintenance is constantly negotiated. Den Hollander [40, 41] also suggests that obsolescence and product lifetime are not just technical qualities, but affected by social, economical and interaction factors that can be influenced and changed over time.

By focusing on the relationships and the social arrangements taking place in software production and the technical constraints shaping development, scholars in science and technology studies contextualize software production not only as a technical issue, but as a socio-technical one, in which technical constraints will shape social order, as much as social and organizational forms will define how software is produced. This can be seen in Kocksch and Jensen recent study of cybersecurity practices in small and medium enterprises [92], places in which fragility must be managed and risk constantly negotiated.

At a more macro level, Maldini et al. discuss how research literature and current regulations on product lifetime extensions are often based on the unproved assumption that industrial production does not come as an adaptation to user demand, but elements tend to suggest that overproduction occurs frequently [93].

1.4 Research methodology

This PhD thesis relies mostly on qualitative research methodology by case study research, involving data such as semi-structured interviews, community and conference ethnography, and documentary data). A detailed methodology description can be found in each of the two case studies of this PhD: for Android in chapter 2 and for Debian in 3.

1.4.1 Semi-structured interviews

The semi-structured interviews that I conducted during the three years of this PhD research play the most important role in it. These were interviews with selected developers, but also activists from civil society organizations, policy and advocacy actors or company executives. They helped shape my research and provided the most important data to analyse.

1.4.2 Community ethnography

During this research I did conference and community ethnography, by following in person several week-long community gatherings in the case of the Debian study, and several multiple day conferences on the mobile subjects in the case of the Android study. During these gatherings, I could identify and perform in person interviews, have informal discussions and participate in community organizing, discussions and working sessions that helped me better understand social and technical issues.

1.4.3 Multi-dimensional online data gathering

Technical exploration of online documents, forums and software techniques were important complementary data in my research. This was made possible by reading technical documentation and scientific literature, following specialized media content, as well as forum discussion, both public or community inner ones. I also followed chat conversations, and mailing list discussions, especially in the case of Debian. Media research and open-source investigation techniques in which I have trained, facilitated this work.

1.4.4 Data analysis

In the Android case study, controversy mapping helped me map the Android ecosystem, its actors and identify points of frictions in my topics of software maintenance and obsolescence. In the Debian case study the data I gathered was analysed using thematic analysis. Transcribing and analysing interviews helped me identify the main topics and issues to analyse further.

1.5 Positionality statement

I am an active member of La Quadrature du Net, a French nonprofit organization defending digital rights, as well as of April, a French nonprofit defending and promoting free software technology and its ethical and social values [21]. The experience I have gained by participating in these organizations, was a great help to me in my thesis: the analytical techniques, data-oriented research methods, and participating or having access to civil society movements proved invaluable. For one year while working on this thesis, I worked as a paid employee one day a week for La Quadrature du Net, as part of my thesis supplementary research activities. In this context, I helped the association set up a working group on environmental and technological issues. In my view, this research work and the community engagements have enriched each other, both through their mutual contributions. They are not always easy to balance, but overall they complement each other, by providing a step back on issues or by offering a different perspective.

I have been continuously installing and testing alternative mobile OSes for the past 10 years, and discussing within the community of hackers problems and solutions related to software on old smartphones. I have also organized workshops on helping users install alternative FOSS operating systems or software, depending on their needs or security threat model.

I am a Debian OS user, and have participated in install party workshops helping people install FOSS Linux systems and software on their personal computers. I have had the opportunity to meet with Debian community members and attend a community gathering prior to my thesis.

[21] See La Quadrature du Net's and April's websites.

In my research team, Limites Numériques [22], my colleagues and co-authors have backgrounds in design and computer science. We collaborate in a broader research project studying digital obsolescence and device longevity from a technical and social point of view. As such, we have followed and organized numerous exploratory workshops and discussions both with developers, hackers and users on software usage, settings, tweaking, or hacking. This has helped me design my research methodology and gather research data. But above all, it has allowed me to engage in rewarding collaborations in an environment that has enriched and broadened my knowledge in new fields such as human-centered design in computer science and human-computer interaction.

[22] https://limitesnumeriques.fr

Chapter 2

Producing software obsolescence: the case of Android OS

Contents

2.1 Introduction

Smartphones appear to be the poster child of "disposable technology", i.e., devices that are neither maintained, nor meant to be repairable or recyclable, but rather replaced [45, 46]. By way of

illustration, more than 1.2 billion smartphones were sold worldwide in 2024, with an average lifespan around 3.5 years. By comparison, embedded software typically have lifespans of 7 to 15 years. And the software on the Voyager 1 space probe software is still running nearly 50 years after its launch, with software updates performed in 2023.

The lack of updates is one of many reasons why consumers renew their smartphones. Although social and psychological factors are at play in renewal decisions, software factors do have a significant role [94]. Without software maintenance, devices tend to become slower and less reliable, but also less secure.

In the Android ecosystem, devices are rarely updated, despite the release of a new version of the Android operating system (OS) every year. The proliferation of Android phone vendors, device models, and active OS versions, known as the Android fragmentation problem, generates maintenance issues at both the OS and application levels [95].

Recent studies define the lifespan of devices in terms of obsolescence [40, 41], viewing it not only as a technical or engineering quality, but also as being influenced by socio-cultural and socio-economic factors. As Cohn argues in her work on software maintenance [89], we hypothesize that the Android update process should also be investigated through a social and organizational lens. Rather than a simple OS, Android can be understood as an ecosystem of organizations involving actors with different values, who interact through code exchanges, shared libraries, documentation, test benches, commercial partnerships, competing or complementary business models, to name but a few.

In order to study the socio-technical challenges associated with the development and maintenance of Android, we carried out a multifaceted fieldwork. We conducted 12 interviews with key players in the Android mobile ecosystem (developers of the Linux kernel, Google, Fairphone, Commown, and alternative mobile OSes or libraries). We supplemented these interviews with conference ethnography, and an analysis of technical literature.

Building upon this corpus, we seek to answer the following questions:

- How is Android structured? Who are the actors involved, and what is the Android building process?
- What inhibits Android updates? Where does software obsolescence manifest itself?
- What are the strategies of actors in Android or non-Android ecosystems in tackling maintenance issues?

After presenting some background and related work on Android and its maintenance, we detail our methodological approach. We then present our results as follows: first, we show how the Android operating system is structured into different software layers, the organizations involved in developing each of these layers, and the detailed pipeline for building the Android system. We then show what inhibits updates, and where obsolescence occurs in practice: which actors are involved and how. We analyze the development flow of the Android ecosystem, how actors interact in building and maintaining (or not). This allows us to identify maintenance breakpoints, that lead to premature end-of-life of devices and, consequently, their obsolescence.

Our findings show that there is not one Android OS, but rather one specific Android build for each smartphone model. The lack of updates appears at the kernel level, i.e., at the core of Android builds, as the code from phone vendors and system on chip manufacturers increasingly diverges from the original Linux kernel code. Google, the main actor governing the Android ecosystem, addresses

maintenance issues by seeking to create a dedicated private space within the OS for industrial actors (phone vendors, system on chip manufacturers), whereas the open-source community would prefer these same actors to contribute and share their knowledge. However, as vendors are the least inclined actors to maintain their code or share their knowledge, the problem remains. Given this tension, we observe how the remediation and maintenance strategies diverge, how the maintenance responsibility shifts from one actor to another, and how, driven by a concern for longevity, some phone vendors and alternative free and open source mobile projects implement remediation strategies to maintain devices.

Drawing on the Android ecosystem and existing literature analyzing complex socio-technical ecosystems involving open source actors, we discuss the values and interests of actors, how they enact power, and the interplay between openness and closure in software development and maintenance.

2.2 Background on Android and related work

Android is the leading operating system worldwide. According to Google, it held 70% of the mobile OS market share in 2023, with approximately 3 billion active Android devices [1], with the remaining 30% mainly held by Apple iOS.

A specificity of Android and its ecosystem is that it is structured around a major technology company, Google, but relies at its core on the Linux kernel, a free and open-source software (FOSS). It also involves industrial smartphone assemblers and manufacturers (known as OEMs, for Original Equipment Manufacturers, sometimes also referred to as "vendors"). This ecosystem is a unique mix of communities and organizations that interact with each other in building and maintaining the system.

In this chapter, we focus on the specific challenges related to maintaining and ensuring the evolution of the Android operating system on smartphones, both old and new. What interests us here is the "dumpster fire" [96] of socio-technical problems at the interface of the operating system, firmware (software embedded in hardware) and hardware, and how the various actors in the Android ecosystem engage (or not) in maintenance efforts.

2.2.1 Maintaining the Android OS and Android applications in a fragmented ecosystem

A major version of Android is released every year, with the latest being Android 16 in June 2025 [2]. These new versions of the Android OS are not distributed or installed on all active devices: at any given time, several Android OS versions are active simultaneously among them. Figure 2.1 shows the distribution of Android OS versions on active devices in April 2025, one month before the release of the new version 16. Android 15 was present on only 4.5% of the devices, versions 10 to 14 (six to two years old, respectively) were all widely used[3], and more than 50% of the market was running an OS that was at least four years old OS (Android 12 and earlier version).

This proliferation of Android OS versions, mostly old ones, installed on many devices at any given time, has been called the Android fragmentation problem. It began being discussed and analyzed

[1] Google blog - Android Updates 2023, accessed on Jan. 29 2025.
[2] Wikipedia - Android version history, last accessed in Sept. 2025.
[3] Android distribution data from Google, April 2025, last accessed in Oct. 2025.

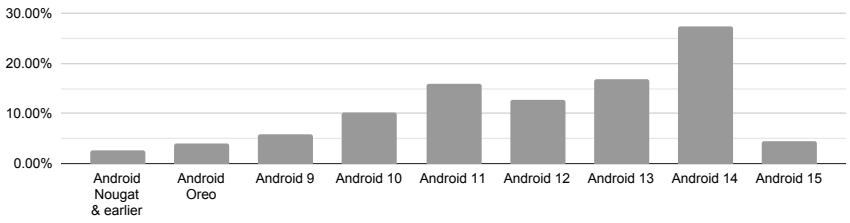


Figure 2.1: Marketshare of Android OS distributions in April 2025.

in specialized media[4] or business reports[5] as soon as 2010. Over time, fragmentation continued being discussed as being at the core of maintenance concerns of both the Android OS and of the applications running on top of it by both Android developers[6] as well as scholars [95, 97, 98].

In 2012, fragmentation is described by Singh, as a two-sided problem: a hardware-based and a software-based fragmentation[7]. This description is further analysed by Han et al.: hardware fragmentation refers to the lack of hardware uniformity on Android—with Android devices holding different System on Chips (SoC), screen types, modems, etc.—which generates a lot of edge cases and bugs for developers to handle; while software fragmentation refers to end-user devices running on different versions of Android OS [97].

At the software level, Android application developers face portability and compatibility issues, meaning that their code does not readily support multiple devices or multiple Android versions (and underlying libraries) [97]. Authors Wei, Liu and Cheung show that app developers have to cope with the ways in which various smartphone vendors defined default values, the way they expose some of their Application Programming Interfaces (API) in idiosyncratic ways, etc [95]. When updated to fit more recent OS versions and to follow the newest Android Software Development Kit (SDK), apps tend to be optimized for newer, more powerful devices. This can lead to performance issues on older phones such as taking a long time to respond, needing more resources, therefore leading to battery use and slow-downs, which in turn lead to faster replacement of devices by users [94]. Linares-Vasquez et al. also note that apps leveraging changing APIs tend to receive lower user ratings [99]. Because App developers are concerned with ratings [100], worsening ratings could lead them to artificially deprecate old applications, to avoid maintenance efforts and bad ratings.

Last but not least, by running old OS versions, phones are exposed to security risks, leading to a high rate of device renewal in work environments, but also in communities and contexts that emphasize software security [101, 102]. This narrative is also pushed by industrial actors who often emphasize security as a reason for OS upgrades, or for changing hardware. Recent scholarship on security brings some nuance. Korn and Wagenknecht propose to consider 'security research' as an ambivalent form of repair and maintenance [103]. They examine how frictions arise in what they call *"social arena of repair"* between industrial actors and hacker and security activists. Kocksch notes that computer security is situated and should be seen as living with fragility [92].

[4] See V. Madhav, Fragmentation in Android: Boon or Bane, 2010, last accessed in Feb. 26, 2025.
[5] Opensignal report on Android fragmentation in 2012, and then yearly until 2015, accessed on Feb. 26, 2025.
[6] Linux World News discussion on fragmentation, accessed in Feb. 2025.
[7] See S. Singh, An Analysis of Android Fragmentation, 2012, last accessed in Feb. 2025.

2.2.2 Smartphones, SoCs and new maintenance vulnerabilities

Android OS has several specific features as an operating system developed for mobile devices. The main component of a smartphone is the system on chip (SoC). A SoC is a single chip containing all the key components of a system soldered side by side: one or more microprocessors, the memory, one or more graphics processors to manage display or AI calculations, modems (for mobile networks or WiFi) and sensors (Bluetooth, infrared, biometrics, etc.). The very fine soldering of these components in a single chip saves space, uses less energy, and allows the creation of portable battery-powered devices such as smartphones. Their design and manufacturing in a single piece allows mass industrial production at lower costs [104]. SoCs are complete embedded systems that began being commercialized in the early 2000. Their technical development has led to the emergence of IoT devices and smartphones as well as companies dedicated to SoC and smartphone production such as Qualcomm, MediaTek, Samsung Exynos.

SoCs come with embedded software which is code that provides low-level control of its components and manages peripheral hardware. When a smartphone is in use, SoC's embedded software manages all the interactions that happen with phone components (touchscreen, cameras, speaker, microphones, etc). A SoC carries both hardware and software architectures, that are co-developed simultaneously during the design flow [105]. This creates new intertwined hardware and software maintenance vulnerabilities: replacing a faulty component in the finely soldered SoC is difficult, even for professional repairers, while embedded software adds a layer of dependency that, according to Greengard *"makes it easier for manufacturers and rights holders to block repairs and control the aftermarket"* [106]. While civil society organizations engage more and more in Right To Repair campaigns[8] insisting on the fact that reclaiming full ownership of hardware and software in order to control devices is an important strategy for addressing obsolescence, there is still a need to better study the role that SoCs play in the obsolescence of smartphones or other embedded systems. Moreover, the IIoT, Industrial Internet of Things, seems to foster obsolescence behaviours, from the higher disposable aspect of the objects being produced and constantly pushed to the market without proper maintenance management, to newer forms of software and cloud-induced operational disruptions which accelerate the process by which these devices become e-waste.

2.3 Methods

My investigation was carried out on three complementary levels. I conducted 12 interviews with actors of the Android OS ecosystem. I also carried out conference ethnography at both in-person and online conferences and developer gatherings of Android ecosystem actors (listed in appendix A). I also analyzed historical and technical documents on specialized websites, media and developer online spaces.

2.3.1 Interviews

I conducted formal semi-structured interviews with twelve informants from September 2023 to November 2024. Some of them took place in person during conferences or gatherings, while others were conducted online. For each interview, I presented the purpose of the interview and explained

[8] See e.g. Right to Repair Europe, PIRG USA, or organizations on the iFixit worldwide map.

how the information would be used. In an interview consent form, informants who agreed to participate could choose to appear anonymously or with their real name and professional activities. One participant preferred anonymity. Furthermore, every participant received a copy of the paper that was submitted, with an invitation to review and discuss what they would consider as misrepresentations or errors.

Informants

#	Name	Organization/Project	Function	Date	Duration
1	Agnès Crepet	Fairphone	Head of Software Longevity & Information Technology	2023-09-14	1h
2	P2 (anonymized)	Google	Developer	Fall 2023	1.5h
3	Marvin Wissfeld	MicroG & LineageOS for microG	Main developper of microG	2023-12-22	1h
4	Emanuele Rocca	Debian & ARM	Debian developer and maintainer, ARM developer	2023-11-25	1h
5	Ben Hutchings	Debian & Linux kernel	Maintainer for Debian Linux kernel packages and linux-firmware repository	2024-05-20	1h
6	Arnaud Ferraris	Mobian	Mobian founder and team leader	2024-11-19	1.5h
7	Federico Ceratto & Jochen Sprickerhof	Mobian	Developers	2024-05-16	1h
8	Denis Carikli & David Ludovino	Replicant OS	Main developers	2024-07-01	2.5h
9	Johannes Schauer Marin Rodrigues	Debian & MNT reform	Developer for MNT Reform port in Debian	2024-05-19	1.5h
10	Elie Assémat	Commown	Cofounder	2023-09-20	1.5h
11	Adrien Montagut-Romans	Commown	In charge of advocacy towards France and EU administrations	2024-09-16	1.5h
12	Simon Gougeon	UnifiedPush	Main developer	2025-12-05	1.5h

Table 2.1: List of interviews conducted during the study of the Android ecosystem.

I interviewed 12 main informants presented in Table 2.1. A significant part of my work consisted of identifying informants with deep knowledge of Android and its ecosystem. Given Google's centrality, together with my thesis supervisor we contacted several developers working at the company while being transparent about our research topics and questions. After several months we received a final email stating that the corporate position was not to communicate with researchers about Android on these sensitive topics. Nevertheless, we interviewed an informant working at the company before we received the "official" position.

At the smartphone manufacturer and vendor level, we conducted interviews with Fairphone employees, a company that builds and markets a "fair and durable" device, as well as Commown, a company promoting smartphone longevity through an original business model.

At the OS level, my study of the Debian OS presented in chapter 3 was very helpful. Having access to the Debian community gave me the opportunity to interview developers working on the Linux kernel upon which Android is built together with interviews of various Debian OS developers. Both clarified distributed development practices related to maintenance.

Notably, I interviewed two developers with experiences on adapting the Linux kernel on a new

System on Chip (SoC), a process called porting. These interviews clarified how the kernel communicates with hardware embedded software (also called firmware) at the SoC level, in order to implement full device functionality at the OS level (through drivers).

I interviewed several developers and community members of Android-based alternative OSes, such as LineageOS, LineageOS for microG and Replicant. I interviewed the main developers of two important tools for the Android OS: UnifiedPush, a decentralized open-source protocol and libraries for push notifications in Android that follows the IETF Web Push standards; and microG, a free open-source implementation of the Google Play Services. I also interviewed non Android, mainline Linux-based OS developers from PostmarketOS and Mobian. These helped me understand community coding practices and policies implemented over time, in order to facilitate development and maintenance in smartphone OSes that derive from Linux, but differ from Android's industrial dominant ecosystem both in coding values and practices.

Analysis process

The interviews were crucial in understanding the Android development ecosystem, actors, interactions and identifying friction points, or points of interest for our research questions. Because they took place at different stages of my research, over an extended period of one and a half year, they played different roles. Some were decisive in that they brought to my attention an unexpected issue that proved important in my understanding. These key issues then guided my next interviews: when something caught my attention during an interview or the analysis process, I organized new interviews and continued conference and desktop research to explore these specific issues and deepen our understanding.

The interviews were audio-recorded with the consent of the interviewees. Most of them were first transcribed to text by using the Vosk offline open-source speech recognition toolkit[9], and then I did a second, more thorough manual transcription. The transcripts served my study in several ways: deepening my understanding, highlighting uncertainties and questions to be clarified, triggering new interviews or research. I kept a text journal of the most important quote excerpts from these transcripts, and what they triggered or highlighted. Many of them also appear in this article. Further quote excerpts were added from conferences that I attended in person or watched online, all being freely available in audio, video or text-transcribed versions. This selection was motivated by the relevance to our mapping of the ecosystem, to the interactions between actors, and finally, to our findings and discussion points. During the writing process, the quotes were carefully checked within their context before being used. At the end of the writing process, they were also checked by quoted informants themselves, that were invited to review them in the context of this written chapter and corresponding article, and make changes if necessary.

Mappings of the interactions within the ecosystem, of the Android stack, and of the development pipelines played an important role in the analysis. The iterative process of creating, discussing, and fine-tuning the resulting figures with my team, enabled to identify limitations in our understanding of Android, map interactions among actors and organizations, and instantiate abstract discourse to specific development activities. These diagrams also enabled us to confront our understanding with external informants familiar with the Android ecosystem. They could (in)validate our understanding of Android, signal elements they discovered thanks to this work, or direct our attention to shortcuts

[9] The Vosk Speech Recognition Toolkit repository

or missing elements in our mappings.

2.3.2 Conference ethnography

I participated in person in several conferences, multiple-day community gatherings and followed numerous online conferences on the technical aspects of Android OS development and update process, including: the annual Linux plumber conference (online), the annual Linux kernel recipes conference (online), the Open Firmware conference (online), Capitole du Libre (online in 2023, in person in 2024), the Free Silicon conference (in person) about free and open-source design and manufacturing of chips, the Debian OS annual conference (online) two European Debian OS community gatherings (in person), as well as the FOSS on Mobile Devices conference day at FOSDEM in 2024 and 2025 (online). A list of the most relevant conferences and talks used in this work can be found in appendix A. This conference immersion also enabled me to conduct interviews in person, get recommendations, and learn through direct discussions with developers from the Linux, Android and alternative OS communities some dynamics that did not surface in more formal communications.

2.3.3 Technical documentation immersion

To complement the ethnographic work, I analyzed the official Android developer documentation by Google, technical documentation on alternative Android OS based systems such as LineageOS, LineageOS for microG, /e/OS, Replicant, GrapheneOS and on alternative non Android mobile OSes based on Linux, such as PostmarketOS and Mobian. I also discussed about these issues with computer scientists, developers, or hackers working directly on Android or smartphone related issues. These helped me better understand specific problems or techniques such as fragmentation, porting, upstreaming, mainlining, backporting. I also followed specialized news and analysis media such as lwn.net (a Linux news and information website), 9to5google.com (news about Google) Android Authority (news about Android) or OS News (news about operating systems), and selected articles from Ars Technica, The Verge or Wired magazines. I also leveraged developer community forums and mailing lists for specific information about OS releases (e.g. xda-developers, Reddit, Fairphone or /e/OS community forums).

Reports and research work by and on the Right to repair movements in different continents [107, 108, 109, 110, 111] together with the interview of participant 11, one of Commown's co-founders, working almost exclusively on advocacy issues, were important to understand software-hardware repair and maintenance issues worldwide, as well as the state of regulations in the US, where the movement for Right to Repair originates, in France, and at the European Union (EU) level. At the economic and legal level, the reports on the Android antitrust infringement cases against Google in Europe [112, 113], in the US [114], in UK [115, 116], recently in Japan[10], and in particular, the detailed investigations on Google that some of these reports detail, helped us to understand Android practices, to confirm or mitigate some of our findings, and to learn more about contractual relations between Google and vendors.

[10] On the Japan Fair Trade Commission's Google Decision: Some Early Reflections, Sangyun Lee, Kyoto University, April 2025, last accessed in Nov. 2025.

2.4 Results

We now outline the socio-technical challenges of developing and maintaining Android. We first present an overview of the Android software layers, focusing on the ones that are relevant to upgrading and maintaining Android. We present the organizations involved in developing these layers, and the Android build pipeline (2.4.1). Building upon this technical picture, we detail how obsolescence happens in practice (2.4.2) and what inhibits maintenance (2.4.3). Simultaneously, we present how the actors involved tackle (or not) maintenance problems (2.4.3), the strategies they develop to enable software maintenance on aging smartphones and their limits (2.4.3).

2.4.1 An overview of Android

As the most widely used operating system in the world, Android should not be seen only in the technical sense assigned to operating systems in the Computer Science literature, i.e. a tool to abstract device hardware and manage resources for its users and their applications (the OS kernel). We study Android as a complex ecosystem, involving hardware manufacturers, phone vendors, app developers, and end-users, all interacting with each other in technical but also commercial, industrial, legal and social ways.

Android software layers and actors.

To give a sense of the Android architecture, in Figure 2.2 (left) we offer a simplified overview of its software layers, and for each of them we explain their role and the actors involved in their development, maintenance and governance.

The **Hardware and Firmware layer** (bottom) represents the code embedded into the hardware of smartphone devices. Every piece of smartphone hardware, be it modems, cameras, sensors, touchscreens or System on Chips (SoC), comes with embedded software, often called firmware. Embedded software is necessary to ensure the functioning of the hardware and to provide low-level control of the hardware to higher-level software such as the operating system. This software is developed and maintained by the hardware manufacturing companies for the Android ecosystem. They are responsible for providing fixes when bugs or security issues are discovered, or for updating the software when a new version of the Android operating system requires it.

The **Operating System (OS)** abstracts device hardware and manages the device resources for its users and their applications [117]. The OS communicates with the hardware via dedicated

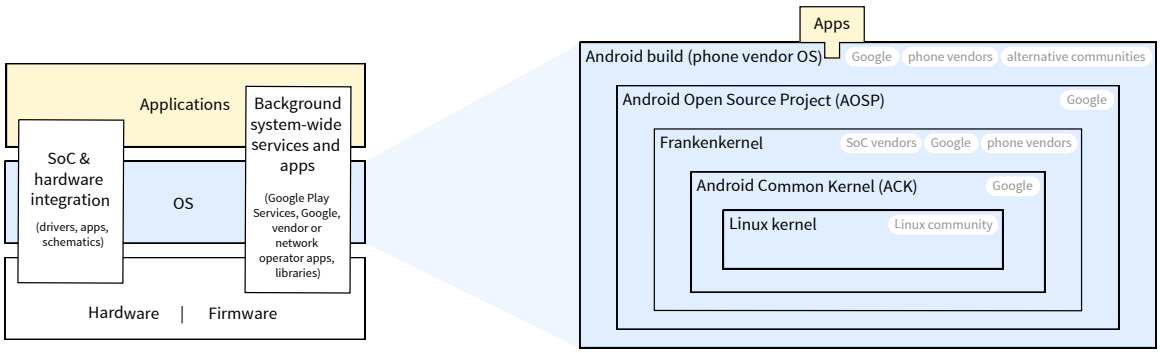


Figure 2.2: Android software layers (left) and Zoom into the Android OS composition (right)

software libraries often called drivers, which interact directly with their firmware. Google is the main developer of the Android OS, defining a general scheme for these drivers and the system to interact with hardware components. The drivers and OS modifications are provided by phone vendors and phone hardware manufacturers in order to ensure that the specific cameras, modems or other components are functional in the OS.

Finally, the **Application layer** is where anyone can introduce new software, with the aim of developing applications (apps) that make the phone directly usable to its users.

On top of these three software layers, we identified two cross-layers. First, the **Background system-wide services and apps** containing applications and libraries that often run in the background (are not immediately visible to the user), are pre-installed in the phone, and have special privileged access to the whole Android system. These can be created by Google, phone vendors or network operators and include services or applications such as localization, data tracking, advertisement injection, push messaging, permissions management, network services, etc. Most of the time they cannot be uninstalled by users, and if blocked or uninstalled by experienced users using special techniques there is a risk of altering phone functionalities. They are most of the time proprietary code software of Google or phone vendors, as for example Google Play Services (a Google system-level package of services that almost all Android apps depend on), Samsung Push Services (a Samsung system-level package of services that Samsung apps depend on), etc.

The second transverse layer is the **System on Chip & hardware integration** one, which we identified as a collection of software code, firmware and drivers, hardware documentation and schematics, or "hacks", that are needed in a mobile phone to support its particular SoC, and all other hardware such as touchscreen, battery, etc. They can be located both in the OS layer for their core functionalities (for example, a generic driver making a camera work), but can also be very specific to the device, or in the application layer (to make the camera perform in a device-specific way, make brighter pictures, have special features, etc.). Without them, the device cannot function properly.

From a Linux kernel to an Android build.

The Android installed on a smartphone is the result of multiple actors: from the Linux community to Google, SoC manufacturers, phone vendors, and network operators. In the following, we detail the various software artifacts involved in the production of Android as it is deployed (see Figure 2.2, right).

An Android build corresponds to a version of Android created for a specific device model. For each smartphone model, a specific Android is built and installed on it, which is ultimately the Android build experienced by end-users. These builds are all based on Android Open Source (AOSP) developed by Google, but diverge in what Marvin Wissfeld (participant 3) explained as follows:

> "*Each phone has its own OS, there is no universal installable image for Android like we are used to in desktop computing. The AOSP is open source, but we cannot deploy it on physical hardware, for that we need additional drivers or firmware, that are available only for the specific hardware the phone is made of, and are not open source in general. A custom ROM, an Android build for a device, is what you put on top of the AOSP plus drivers and firmware strictly needed to get the hardware running, and many of the services and applications that Google and manufacturers put on top of it. AOSP itself is not for real hardware, is not relevant to end users, it is running only on emulators.*"

At its core, Android is based on the free and open-source Linux kernel. A kernel is a basic fundamental software component at the core of an OS, providing important hardware functionalities and facilitating hardware and software interactions. Over 1000 contributors forming the Linux community contribute to each release, which involves over 8 million lines of code. Every 9 to 10 weeks a new stable mainline kernel is released and published on kernel.org. Usually once a year, a stable kernel is picked and designated as a long term support (LTS) kernel.

Google develops and releases a new version of AOSP every year. To do this, an LTS Linux kernel will be used by Google to create the Android Common Kernel (ACK) at the core of every new AOSP version to be released. An ACK is an LTS Linux kernel with extra code from other branches of the Linux kernel such as *"new Android features under development in the Linux community*", and "*Vendor/OEM*[11] *features that are useful for other ecosystem partners*"[12]. Any version of AOSP contains an ACK and all the necessary functionalities of an OS that make its specificity: in the case of Android the way the system interacts with hardware components (e.g. the touchscreen or the camera), how hardware components interact with each other (e.g. the battery with the CPU), how the user interacts with the system (e.g. through a custom UI layer).

Luca Weiss, an Android developer at Fairphone, describes how the ACK is then used[13]: " *Based on [an] ACK branch, the SoC manufacturers take it, add some support for their SoC on top, and then finally, device manufacturers get this code base and put their device-specific changes on top*". First SoC manufacturers develop their kernel by adding a large amount of SoC specific code on top of an ACK kernel, this ACK corresponding to a given Android AOSP version and a given LTS Linux kernel. For every device manufactured with a given SoC, vendors take this SoC's specific kernel and add to it the extra hardware-specific code and drivers (for the touchscreen, battery, camera, etc.). The resulting kernel is what is sometimes called a *vendor kernel*, or sometimes, "*frankenkernel*"—a term that we will discuss further below.

On top of the vendor kernel, vendors take the corresponding AOSP version released by Google, usually add to it drivers needed by the phone components, creating an Android OS build able to run only on this specific phone device. In this phase, vendors also customize Android by adding their own user interfaces, system-wide services or apps as part of the transversal Background layer. When these phones are marketed by network providers, the latter also customize the Android system by adding their own user interface, system-wide services or applications. most of the time these changes (drivers, services and applications) are not open-source code.

The specific nature of the Android OS, consisting of one build per device, and how these builds accumulate specific software from various actors, is one of the first main findings of our study. We noticed that this is not something that is widely known or understood, even among developers or on specialized media. From a software perspective, smartphones are indeed quite different from personal computers, where the same generic OS (whether based on Linux or Windows) can be installed by the user regardless of specific key hardware components such as the motherboard, the processors, or the memory they hold. In Android smartphones, the SoCs, holding both key hardware and software components, play a central role in the specificity of the OS builds, on top of which actors each add their own software layer of specific hardware features, services or applications.

[11] OEM: Original Equipment Manufacturer
[12] From the AOSP official documentation, accessed on Jan. 13, 2025.
[13] Mainline Linux on Fairphone? Yes, please!, Capitole du Libre, Toulouse, Nov. 2023, last accessed in Dec. 2025.

2.4.2 Obsolescence in action

Given the Android build pipeline described in the previous section, we will now clarify how specific Android builds pose maintenance issues and can become obsolete in the ecosystem. In Figure 2.3, we illustrate this build process, from the Linux kernel to Android builds, for two specific smartphones: the Motorola Moto G7 and the Fairphone 3.

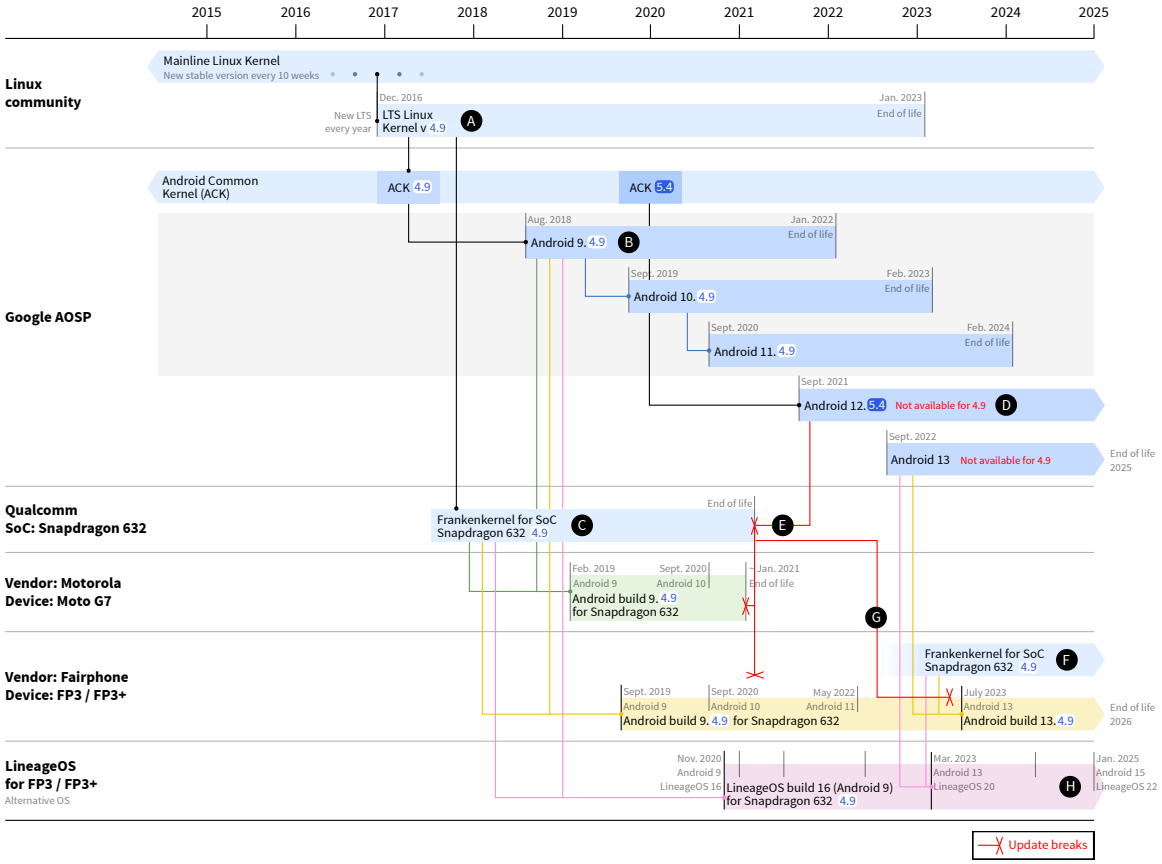


Figure 2.3: Android builds for Fairphone 3 and Moto G7 phones with update breaks appearing at different points

How lack of updates appears.

Every year, Google releases a new version of Android based on two or three of the latest Linux LTS kernels and creates an ACK for each LTS kernel used. Each ACK will be the core of the new Android versions specific to a device. For example, Google Android OS version 9 was released both as version 9-4.4 (based on the ACK coming from LTS 4.4), and as version 9-4.9 (based on LTS 4.9). As we will explain later, maintaining or disrupting maintenance for one of these versions has an impact on the obsolescence of the devices that were put on the market with these Android versions. In Figure 2.3 we only detail the Android development process based on LTS 4.9 (marker A).

The Motorola Moto G7 and the Fairphone 3 were both put on the market in 2019, and were both shipped with the same Qualcomm Snapdragon 632 SoC. Both smartphones were running on

Google's Android version 9-4.9 (marker B), and contained a kernel built by Qualcomm specifically for the Snapdragon 632 SoC in 2019 (marker C). The LTS kernel 4.9 of these Android builds, was released in 2016 by the Linux kernel development community.

When they were released, in 2019, the Android builds on both smartphones were based on a 3 years old Linux kernel. The LTS 4.9 had been released in 2016, and the Linux community defined its "end-of-life" (end of official support) in January 2023. According to A. Ferraris (participant 6), Google needs time to develop an ACK based on a given LTS kernel (one or two years), after which SoC manufacturers in turn need extra time to develop the SoC's specific kernel on top of this ACK (another one or two years).

Qualcomm, **the SoC manufacturer, never upgraded its kernel after the 4.9 LTS kernel**. Qualcomm only applied some of the security updates that LTS 4.9 received from the Linux community. Qualcomm applied these updates to its SoC kernel for only two years before stopping updates altogether in 2021. Motorola then applied these updates to its Moto G7 phone before stopping in 2021, two years after the phone's release. At that time, the Qualcomm frankenkernel was based on a five-year-old Linux kernel that had been upgraded five times to newer versions, each of which had been updated many times with bug fixes or security patches.

These two observations reveal poor maintenance practices in the Android ecosystem: few software updates and early end of support. This is quite unusual when compared to development and maintenance practices of some of the most well-known operating systems derived from the Linux kernel in the Desktop world, such as Debian, Ubuntu, Fedora, Red Hat OS, etc. They generally closely follow the updates and upgrades of the latest LTS kernels, by implementing all bug fixes and security patches, as well as new features and developments contributed to the Linux kernel as hardware evolves. Study participants emphasized how alignment with LTS kernel releases facilitates frequent but small changes, as opposed to larger, more complicated changes, when updates are infrequent and take place after long periods. Luca Weiss from Fairphone explains [14] when talking about Qualcomm's frankenkernel 4.9 for Snapdragon 632:

> " *The code that we got from Qualcomm [of frankenkernel 4.9] is about 2.5 million lines of difference compared to the 4.9 LTS kernel that is being released upstream. And it is about 18000 commits. This also shows why it is not really possible for a device manufacturer to rebase all of this 2.5 million lines and make them work.*"

"Rebasing" is a development practice consisting in applying changes originally made to a derived code (in this case the frankenkernels), to the original code from which this code derives (here the LTS Linux kernel). This implies solving code conflicts and malfunctions that the rebasing may introduce, while making sure everything works how it should. The biggest the changes between the derived code and the origin are, the more difficult the rebase becomes.

The changes made by Google to the LTS Linux kernel are also significant and not aligned with the LTS kernel, as explained by Arnaud Ferraris from the Debian and the Mobian mobile project:

> " *When Google makes the kernel for Android [ACK], it modifies the whole core part of the Linux kernel with the scheduler, memory management, power management, to make it compatible with the Android OS. Google tries to upstream this code so that it is available in the Linux kernel, so when the Linux kernel gets updated Google does not have to*

[14] See 13.

deal with it anymore, unless Google changes these parts. But the process of upstreaming patches into the kernel takes time: there will always be a first version, comments from the Linux kernel team, things that the Google developers had not thought of—the kernel maintainer will have their own Linux logic—and code will generally need to be modified several times before we gradually arrive at a final version. So, despite a genuine desire to do things upstream, we still have an ACK from Google that has a lot of downstream code when it is passed on to the SoC manufacturer. But, downstream code is code maintained only by Google. Every time the Linux kernel is updated, the downstream code must also be updated from Google, and then passed to SoC manufacturers once again. This code may become upstream at some point in the future. But at a given time, it is downstream code. Why? Because upstreaming takes time, and Google's pace is not that of the Linux community, they are not completely in sync, they have their own Android pace."

Upgrades lacking backward compatibility.

Beside the lack of software update previously observed, we observe new breaking points that appear when Google upgrades, on a yearly basis, from one version of Android to another.

As illustrated in Figure 2.3 (marker D), in 2021, two years after the release of the Moto G7 and Fairphone 3, **Google introduced kernel backward incompatibility in the new Android version 12, which was based on kernel ACK 5.4, and no longer on ACK 4.9.**. Android OS upgrades were no longer available for 4.9 kernels. This marked the end of the support from Qualcomm of its 4.9 frankenkernel for the Moto G7, the Fairphone 3 and all other phones holding this SoC (which was confirmed by P1). In 2021, Motorola had upgraded the Moto G7 only once, from Android 9 to version 10[15]. When Google stopped providing the 4.9 version when releasing Android 12, Qualcomm stopped updating the phone's frankenkernel to the ACK of Android 12 and Motorola stopped upgrading the phone to newer Android versions (marker E).

But this backward incompatibility did not make upgrades impossible. As Agnès Crepet (participant 1), head of the *Software longevity and IT* team at Fairphone told us, they managed to upgrade its OS on Fairphone 3, despite the lack of support. When Google released Android 12, and Qualcomm stopped updating the SoC's frankenkernel, Fairphone maintainers took over. Luca Weiss, who was then a volunteer developer on the PostmarketOS project, an alternative community-driven mobile OS, was hired in the team because of his knowledge of the kernel and of the mainlining process. The updating process of the abandoned Qualcomm SoC frankenkernel took time and was finalized after Android 13 was released, so the company decided to release an upgrade of Fairphone 3 directly to this new Android 13 in 2022 (marker F). This update was possible in great part because of the community work from the alternative OSes—PostmarketOS and LineageOS mostly—that had taken over maintenance work on Qualcomm's abandoned frankenkernel by applying security patches and functionalities needed to keep compatibility to the Linux kernel or Android newer versions whenever possible. Their work is made available to the community in open-source repositories, while device trees—data structures describing all the device components needed by the kernel to use and manage those components—are shared together with available documentation. Various discussion spaces for users and developers to interact also exist[16]. Based on this work, the newly formed Software

[15] See Motorola's update announcement followed by specialized media Techradar analysis, December 2020, archived, last accessed in Dec. 2025.

[16] See the LineageOS user and developer wiki, the blog that announces the new development releases and discusses

Longevity team of seven people at Fairphone succeeded in updating an Android build after both Google and the SoC manufacturer had stopped doing so.

Upgrades introducing new features that break compatibility.

One year later, the release of Android 14 brought new breakdowns in updates. As Weiss explains [17]: *"With Android 14 released in 2023, Google is introducing some new features that are not present at all in kernel 4.9. AOSP is dropping support for these kernel versions sort of aggressively, as they do not support any of the new features that [Google] wants to use in Android 14"*. In July 2024, Fairphone announced to its users [18] that the FP3 phone would not be upgraded to Android 14 and beyond anymore (marker G):

> "*We invested considerable time and resources into exploring ways to integrate the new Android system with the existing kernel, and even contemplated upgrading the Linux kernel itself. We also engaged in discussions with Google Android Engineering. The legacy Linux kernel (4.9) used in the Fairphone 3 [...] would not support Android 14 at all.*"

FP3 would thus end software support in 2026, but would nevertheless continue to receive security updates on Android 13 until then. With 7 years OS support, this makes the Fairphone 3 one of the longest supported Google certified Android OS phones, and as some specialized media say, the only phone manufacturer that maintains the OS after SoC support shut down[19].

We noticed that the LineageOS community managed to circumvent the above incompatibilities and keep the Fairphone 3 updated after Qualcomm, Google, and even Fairphone stopped supporting it. In November 2025 LineageOS with Android 15 could still run on Fairphone 3 (marker H). We discuss the circumvention strategies allowing longer maintenance from both Fairphone and LineageOS, but also their limits, in section 2.4.3 below.

2.4.3 Disruptions to maintenance practices in the Android code flow

Based on our interviews, conferences and documentation analysis, we identified issues related to how changes to code developed by each actor are propagated within the Android ecosystem. The layers in the Android operating system shown previously in Figure 2.2, appear as code flows circulating from one actor to another. The breaking points, which we described previously, occur at different levels and affect code maintenance differently. Figure 2.4 illustrates these code flows, or lack thereof, and the resulting maintenance breakpoints.

Divergent maintenance practices inhibiting updates and creating technical debt.

In our interview, Arnaud Ferraris (participant 6) from the Mobian project, a Linux derived OS for mobile, talks about the great amount of added code that makes maintaining difficult in Android:

changes. The same can be found for PostmarketOS, and LineageOS based OSes that maintain Fairphone 3 such as LineageOS for microG or /e/OS.

[17] See 13.

[18] Announcement of end of support for Fairphone 3 in July 2024, last accessed in Dec. 2025.

[19] See *Fairphone 3 gets seven years of updates besting every other OEM*, Ars Technica, July 2023, last accessed in Jan. 2025.

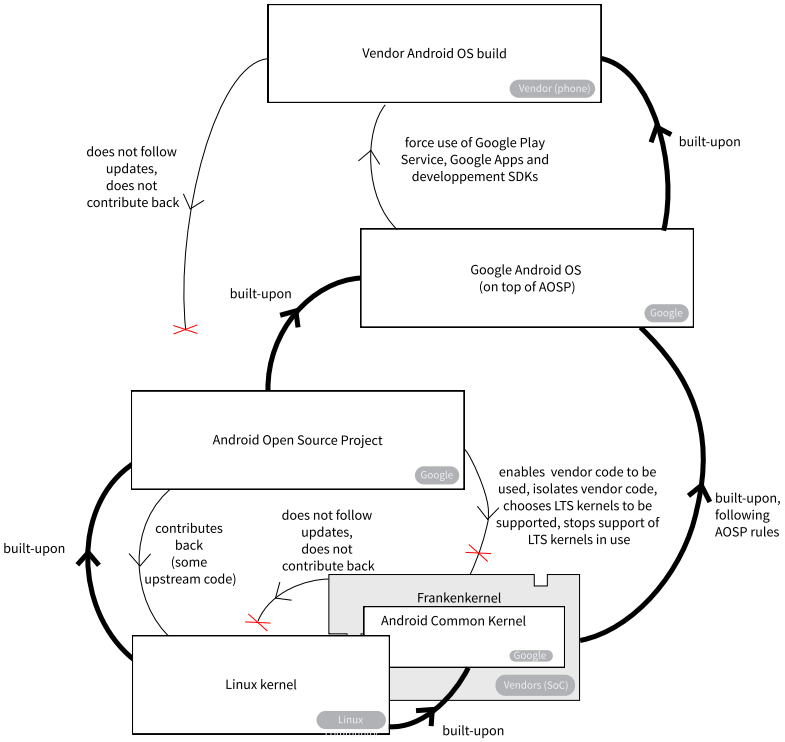


Figure 2.4: The development flow of an Android build, red crosses indicates a lack of contribution to the original code-base.

> "*This ACK by Google with downstream code in it passes into the hands of the SoC manufacturer, they add their own system drivers to manage hardware, then it is passed on to the phone manufacturer who will add other drivers. So we already have three levels of forked kernels, and when we finally look at the cumulative amount of changes that this represents, it's huge. If I take the example of the OnePlus 6 phone—a phone we support in Mobian using a community-developed mainline-based kernel—this was over 5 million lines of code compared to the base Linux kernel used.*"

In free and open source software development, *mainlining* is an important wide-spread coding and maintenance practice that refers to the integration of new developments into the main *mainline* source code branch in the project's repository. When developing upon the Linux kernel, mainlining is a two sided development flow: (1) systems derived from Linux follow the mainline Linux kernel development, and also (2) contribute back code to the Linux kernel as often as possible, through a process known as *upstreaming*.

Ferraris further explains the different development practices between phone manufacturers and free software communities:

> "*There is a big difference most of the time between the kernels from phone or SoC manufacturers and kernels maintained by a free software community: manufacturers will create a software mess that suits their needs, while free software communities will make sure to make the minimum changes necessary for it to work and be accepted into the upstream*

> *Linux kernel. When developing we always try to keep this notion of upstreaming in mind [...] for the sake of sustainability and for the comfort of everyone, both users and developers, it's better for the changes to be upstream, and we try to think in advance how to make the changes we need so that they can be accepted in the kernel.*"

Upstreaming enables mobile OS actors following the mainline Linux kernel, such as PostmarketOS and Mobian, to directly benefit from the various contributions to the Linux kernel. It also enables kernel maintainers to handle upstreamed contributions in manageable bits, isolating problems, using simpler tests and dealing with them on a daily basis. Other Android based OS actors, such as LineageOS, GrapheneOS, Replicant, even if they do not mainline the Linux kernel, follow upstreaming practices as much as they can, and collaborate in understanding and developing free drivers for phone components for which the code is neither public nor documented.

As illustrated in 2.4.2, in the Android ecosystem, Google creates the ACK from the Linux kernel without mainlining it and only partially upstreaming code into it. SoC and phone manufacturers use these ACK to create the frankenkernels by adding code that is not designed to be contributed back to the Linux community: assembled from various internal codes, designed to meet their internal development needs and undocumented for external use, hence the prefix "franken". As the Linux mainline kernel evolves, the difference with SoC frankenkernels continues to grow, making it increasingly challenging to share and propagate code between them over time, as Ferraris explains:

> "*It is code that is written internally, never submitted to the community. A manufacturer will have dozens of very different hardware references and generally, almost always, wants to have a single code base for all of them. As a result, they develop extremely complicated code filled with switches, if statements, internal compilation options used to enable or disable certain parts depending on the specific hardware they're enabling for a specific build. What's more, there is no public documentation for any of this code. Sometimes things seem magical when you read the code. It makes sense to manufacturers, but when we read it, because we don't have any documentation, we don't understand at all why they're doing it, and we wouldn't be able to explain it to the Linux kernel maintainer, or make it work, let alone develop an upstream-compatible Linux kernel driver with it. This also makes reverse engineering complex.*"

The use of nested if statements and switches is a known cause of what in computer programming is called *spaghetti code*, code that is fragmented, entangled, and thus difficult to understand and to maintain [118].

The divergence in development strategies outlined above, has created historical tensions in maintenance practices between FOSS communities such as the Linux kernel community, and hardware manufacturers such as phone vendors in our case. The Linux community aims to build a kernel that runs on as much hardware as possible and for as long as possible, whether that hardware is old or new. Manufacturers focus on developing code for new hardware by adding to their own existing code base. By using the Linux kernel, which implements the core functionalities of operating systems, they reduce the costs and time required for launching them [119]. But they show little interest in maintaining this code or contributing code back to the kernel in order to keep their hardware updated over time. The responsibility for code maintenance is transferred from manufacturers to the Linux kernel community or to the FOSS mobile communities maintaining old hardware. This shifted

maintenance constitutes what is called technical debt in software development [120]. In software using the Linux kernel, the lack or delay of upstreaming, as well as obfuscated and undocumented code, are common factor in technical debt [121].

Maintenance barriers: software obfuscation and circumvention strategies.

As maintenance shifts from vendors to the FOSS communities, access to hardware specifications and documentation becomes important. Luca Weiss, Android developer at Fairphone, explained during a public conference, how access to private documentation shared by Qualcomm exclusively with phone vendors, enables to maintain SoCs, mainlining them with the Linux kernel, rewriting proprietary drivers in open source, and share them with the open source community:

> "*Having access to secret documents now that I am working at Fairphone definitely makes things easier. Schematics, data sheets and documentation are confidential material. Schematics make it easier to build the device trees, or hardware descriptions that the kernel needs, because some things are not obvious from the way the vendor-provided kernel has this written. Data sheets make it easier to write new drivers in case it's necessary.*"

Additionally, hardware and smartphone vendors place a great emphasis on keeping code private whenever possible. The copyleft GNU General Public Licence (GPL) of the Linux kernel requires manufacturers to provide the code from the frankenkernels under GPL-compliant licences. As mentioned previously, this code is provided by vendors, but being too chunky, complex and undocumented, it is of little or no help for the open-source community in maintaining it. For phone components source code is never available, vendor drivers being almost always proprietary closed code. As articulated by participant 2, cameras for instance play an important role in vendor marketing strategies: differentiating devices on camera performance and features are key selling points in the smartphone market. Since software plays a key role in camera performance and their distinguishing features, vendors keep it private and well protected. Ferraris confirms:

> "*Beyond the SoC in every phone there will be a screen that will need a very specific driver, the same goes for touch panels, and cameras are yet another problem because there is a lot of intellectual property involved in them. In the manufacturers kernels there is nothing to be found about these, everything is in the Android user space or elsewhere, in proprietary binary code form. How it works is well hidden.*"

Because of this opaque nature, the binary code and drivers coming from manufacturers is often called *blobs* among Linux kernel developers and open-source communities. Trying to have the less blobs possible, by replacing them with functional open-source code, is an important issue, as these blobs do complicate maintenance when changes occur in the OS or in the kernels, or when security issues involving them arise.

A tension exists between Google and the Linux kernel community on these code blobs. Google attempts to create safe spaces for vendors directly into the Linux kernel structure, and to remove part or all of them from its Android layers. Linux kernel developers that place great emphasis on open source code are reluctant to create these safe spaces for vendors to put their binary code. They fear that instead of pushing vendors to open-up their code or at least contribute to the kernel community with documentation, these safe-spaces will push them to continue non-transparent development practices, which are precisely what complicates their maintenance and updating work.

This tension is clearly illustrated in a meeting report from a two-days workshop within the Linux kernel team dealing with camera hardware [122] involving kernel developers and Google employees. The report shows that Google contributes back, upstreaming code to the Linux kernel camera modules. This is useful both for the Linux kernel and for Google in the development of Android and Chrome OS, both based on Linux. During the meeting Google tries to push its idea to create a vendor specific code space inside the Linux kernel camera framework, where vendors can upstream blobs and keep the intellectual property of their devices safe. The reaction of the Linux kernel team maintainers clearly illustrates the tension: [122]: *"Upstreaming a driver requires opening up the driver interfaces [in Linux kernel practices]. There appears to be near-unanimous consensus on this apart from Google. It is not an option [for the Linux community] to upstream a driver that has support for undocumented closed features. Basically [Linux kernel] maintainers can not put their name on something that contains unverifiable (for them) and unusable (by all except the vendor) features."*

Extremely large commits, spaghetti code and blobs are considered anti-patterns that negatively impact program comprehension [118], specifically during maintaining tasks [123], while program comprehension and documentation are considered central to effective software evolution and maintenance practices in software engineering studies [124, 125]. As illustrated above, all of these patterns appear in the Android OS ecosystem.

Google's two-tier Android development strategy: reinforcing fragmentation when trying to address it.

Google has recognized the persistent Android OS fragmentation problem, (i.e. multiple versions of the Android operating system present on active devices at any given moment), with multiple active Android devices not being upgraded to the latest Android OS version. The company has implemented several strategies to facilitate updates in order to reduce this fragmentation. In 2017, when launching project Treble[20], Google Android development team declares: *"we've consistently heard from our device-maker partners that updating existing devices to a new version of Android is incredibly time consuming and costly"*. Treble consists in implementing a modular base for the low-level architecture of the OS in order to better isolate vendor code. Later in 2020, Google also launches the Generic Kernel Image (GKI) project to address kernel fragmentation by *"moving SoC and board support out of the core kernel into loadable vendor modules so they can be updated independently."*[21].

By isolating vendor code both in the ACK and in AOSP, Treble and GKI attempt to help Android actors offer faster roll-out updates. This technical isolation is welcomed from vendors willing to maintain their phones, as Agnès Crepet from Fairphone (participant 1) told us. But it does not enforce changes to their updating policies and coding practices. Regardless of the underlying technical stack improvement, the obfuscated vendor code remains, and updates still depend on the willingness of vendors in maintaining their code and devices. Data from 2025 attests that fragmentation is still pervasive (see Figure 2.1) and Android OS updates have not significantly improved on user devices since 2017.

The vendor code isolation also facilitates Google's own work of developing and maintaining AOSP as vendor blobs interfere less with the rest of Android. But the approach here differs from that of

[20] Here comes Treble, 2017, Google Developers blog, last accessed in Feb. 2026.
[21] Generic Kernel Image documentation, last accessed in Feb. 2026.

the Linux kernel community on blobs that consists in integrating them as much as possible in the system, in order to facilitate maintenance. Google has no need for vendor blobs to be completely open-source, documented or intelligible. By isolating vendor space, Google can continue developing AOSP while leaving its partner vendors free to "blobify" and maintain secrecy over their specific hardware code.

This allows for another shift in maintenance responsibility, from Google to vendors. But vendors' lack of maintenance remains unaddressed. And it is precisely the lack of maintenance of vendor code—Android OS builds specific to every device that are being little or not maintained—that leads to the Android fragmentation situation.

Google's double-standard Android updates strategy: from deprecation to forced dependencies.

In the Applications and Background services of Android (Figure 2.2), updates are fine-tuned to serve Google's business model and interests. As Marvin Wissfeld (participant 3), that develops microG, a free open-source implementation of Google Play Services, notes:

> "*[Google] stopped updating apps on AOSP, the clock app, the messaging app. They update only in the sense that they make sure they are still compatible, but do not add new features. They fully update only the proprietary versions coming with the Google-licensed Android. In the Google messaging app they integrated their own messaging proprietary system. They leave AOSP behind, add small features to the Google versions of the apps, and put on top of them a dependency on Google Play Services. ... [Manufacturers] take the proprietary Google apps requiring Google Play Services.*"

As an AOSP derived OS, LineageOS development closely follows that of Android: when Google releases a new Android version, LineageOS releases a new complying version. Google's lack of updates in AOSP are quite silent, as official Android release announcements do not usually mention them. By analysing LineageOS new version release announcements (called changelogs), one can better spot this lack of maintenance and sometimes complete deprecation of essential Android apps. For example, in the Android 14 release in 2023, Google deprecated the Dialer and the Messaging applications in AOSP, with a discrete message revealed by specialized media[22], written in the repository of the source code for the deprecated apps, saying: *"This app is not actively supported and the source is only available as a reference. This project will be removed from the source manifest sometime in the future*".

In this case, a shift of maintenance towards open-source communities is clearly at play. When the corresponding LineageOS (version 21) was released some months later, it announced that it had taken over the maintenance and further development of the deprecated apps:[23]:

> "*Since AOSP deprecated the Dialer, we have taken over the code base and did heavy cleanups, updating to newer standards (AndroidX) and redesigning[...]. While Messaging was also deprecated by AOSP, at least the Contacts app was not. Nonetheless we gave*

[22] Google further guts the AOSP by deprecating the dialer and messaging apps, OSnews, June 13, 2023, last accessed in Dec. 2026.

[23] LineageOS version 21 announcing new Android 14 release in Changelog 28, February 14n 2024 last accessed in Jan. 2026.

both of them an overhaul and made them also follow the system colors and look more integrated."

Google's strategy of selective updates in AOSP is accompanied by contractual agreements between Google and its vendor partners. These agreements ensure that vendors systematically install an Android version with Google proprietary services and applications. Such contracts frequently stipulate the exclusive use of Google applications, for instance by prohibiting vendors from installing another search engine. In some instances, the contracts may also preclude vendors from marketing devices using an alternative Android-based operating system. As elaborated in 2.5.3, these practices have been the subject of lawsuits and rulings against Google for abuse of dominant position in several countries worldwide.

As Wissfeld notes, Google Play Services integrate telemetry, gathering user data used for advertisement purpose, advertisement that is injected back in the applications via these same services[24]. The Google Play Services also serve Google's business interests as one of the dominant actors of web and mobile advertisement[25].

How forced dependencies inhibit sustainable maintenance and resilience.

When the use of Google applications and services essential to the Android OS is not contractually enforced, it is *de facto* ensured. Wissfeld (participant 3) told us that he has seen these services evolve and become more and more present in the Android ecosystem. While essential system apps get less updates in the AOSP version, they gain Google Play Services dependency in their final official Android build.

Indeed, the most widely used Android development environments and libraries, such as Android Studio or the Firebase platform, both developed by Google, integrate unavoidable dependencies to Google Play Services and Google's Firebase Cloud Messaging. As a result, most Android application developers integrate these services in their apps. Google's Firebase Cloud Messaging (FCM), formerly known as Google Cloud Messaging (GCM), implements the push notification system that enables applications, such as messaging or emailing ones, to use Google servers for sending notification messages to the apps on the user phones (e.g. each time a new message arrives). Given that one of the essential uses of a smartphone is messaging, the push messaging systems are crucial in mobile ecosystems. The technical implementation of FCM is dependant on Google Play Services: in order for FCM to function, Google Play Services have to be present in the user's phone. However, neither FCM nor Google Play Services are implemented in AOSP. As a result, most vendors market Android phones with Google apps and Play Services, and almost all Android apps integrate the Play Services and use the Google FCM system.

Moreover, the Google Android device certification system, the *Play Protect Certification*, a set of tools for testing Android devices offered by Google to phone vendors and users in order to label their Android phone as *"certified by Google"*, includes mandatory checks of the presence and activation of Google Play Services. Mandatory for device vendors having a contractual agreement with Google, this certification mixes AOSP compatibility tests with Google services and apps tests. Moreover, it is marketed by Google as a safety guarantee for Android phones, while its absence is presented as a

[24]See Ads Safety, the user data telemetry and advertisement injection service integrated in Google Play Services, last accessed in January 2026.

[25]See Google Ads program for mobile apps, Google.com, last accessed in January, 2026.

severe security issue in Google's communication[26]:

> "*Devices that aren't Play Protect certified may not be secure [...] may not get Android system updates or app updates. Google apps on devices that aren't Play Protect certified aren't licensed and aren't real Google apps. Apps and features on devices without Play Protect certification may not work correctly. Data on devices without Play Protect certification may not have a secure backup.*"

Thus, certification becomes another way of enforcing Google dependencies in practice, and presenting their lack as a severe security issue.

In alternative Android-based OSes such as LineageOS, LineageOS for microG, or /e/OS (a LineageOS derivative OS), dependencies from Google are partially removed via use of microG, that reimplements the Google Play Services, removes telemetry and advertisement, but still allows installed apps to use the Google servers via FCM. Open and standardized alternatives that allow to remove all Google dependencies and keep functional systems have been developed in recent years. One of them is UnifiedPush, an open-source system for push notifications in mobile ecosystems implementing the Web Push IETF open standards (RFC 8030, RFC 8291 and RFC 8292)[27], that has been implemented by various open-source systems such as Element, Conversations, Nextcloud, many Mastodon clients, and some browsers based on Firefox. Simon Gougeon (participant 12), creator and main developer of UnifiedPush, explained to us how using an open standard for push notifications offers resilience in the long term:

> "*The Google push notification system is centralized, fully depends on Google. Huaiwei in China has its own centralized push notification system. In countries or situations where Google, Huaiwei, or other actors are not present, or their presence can change for economical or geopolitical reasons, UnifiedPush can provide standardised push notifications for Android phones. Also, in a context of climate crisis, if a region loses its access to the Internet, Google's push notification system is not available anymore. If a regional network takes is implemented until the global Internet access is restored, UnifiedPush can quickly be deployed in it and help keep phones connected. This is the advantage of open decentralized standards, they have resilience and can take over when centralized services are not available, not desirable or have failed.*"

Following our interview, Gougeon published a retrospective article on UnifiedPush, developing on the issues he had discussed with us[28]: *"When a service is controlled by a single entity, nothing can be done when they consider your device too old to be supported*". For him, services as fundamental as push notifications in mobile systems should be implemented inside the main Android code as open standard APIs. Reflecting on the future of UnifiedPush he says *"The best thing that could happen to UnifiedPush on Android [...] would be for it to no longer exist. If Google gives us in Android a system API to let the user define their push service we would not need UnifiedPush anymore. [...] Hopefully, working on UnifiedPush can push in that direction by increasing the demand, and highlighting the need*".

[26] Google Android Help Center, last accessed in Dec, 2025

[27] IETF: International Engineering Task Force, the main international technical standards organization for the Internet. See IETF Web Push push notifications standard documents, last accessed in Jan. 2026.

[28] Simon Gougeon, 5 years of UnifiedPush, F-Droid.org, Jan. 8, 2026, last accessed in Feb. 2026.

imits of obsolescence circumvention strategies]Successes and l

2.4.4 Successes and limits of obsolescence circumvention strategies

imits of obsolescence circumvention strategies

The obsolescence circumvention strategies and maintenance efforts do not come without drawbacks for the FOSS communities and some phone vendors that put effort in it. Fairphone and LineageOS achieved to maintain the Fairphone 3 after Qualcomm abandoned its frankenkernel and Google removed support for Android OS 12, but faced difficulties in implementing these update strategies on the long run.

Update problems became particularly important with the release of Android 12 in 2021, which broke the backward compatibility with kernel 4.9 and below. One reason is that *eBPF*, a tool to manage network traffic in the Android kernels, replaced the old *iptables*. Updating devices running on kernel versions 4.9 *"proved challenging due to the sheer number of commits and structure changes"*, while for devices using older kernels the upgrade was not possible anymore[29].

The update process of taking parts from a newer version of a software system and porting them to an older version of the same software is called *backporting*. It is a common practice when preferred maintenance solutions like *upstreaming* or *mainlining*, are not taking place. Backport code is often applied as patches in order to incorporate changes into an old code-base (the term *patching* is also used).

To maintain the Fairphone 3, Fairphone and LineageOS backported changes introduced in Android 12 to its old frankenkernel (based on kernel 4.9), which both Qualcomm and Google had stopped supporting. They also included security patches from the Linux kernel team, which Qualcomm had stopped offering. After the release of Android 15, Fairphone announced[30] that it would stop offering Android upgrades to Fairphone 3, due to the eBPF structural changes that became too complicated to backport. Yet LineageOS with Android 15 can still run on the Fairphone 3, thanks to backports provided by the open-source communities, even after Fairphone stopped its upgrades. But, for vendors aiming at the Google certified phones, like the Fairphone does, the stricter rules and controls of Google's *Play Protect Certification*, make things more complicated than for alternative actors, used to release OSes with sometime bugs and missing features, and to fix them in subsequent updates when possible.

Nevertheless, as Google adds new features with each Android release, backports become increasingly difficult to maintain. For developers in the open-source communities we interviewed, backports are seen as a temporary and inadequate solutions to the update problem. Because backports are code that is neither mainlined nor upstreamed, but comes as patches of code applied at a given moment, when the ecosystem evolves, patches need to be updated independently every time. Thus, patches face the same update issues as frankenkernels from vendors do: they are difficult to maintain, especially when they come in big chunks of code.

Some alternative mobile OS projects such as PostmarketOS or Mobian, choose to completely detach themselves from Android and its frankenkernels by directly following the Linux mainline kernel development for mobile phones. Older devices are regularly abandoned by LineageOS maintainers,

[29] LineageOS 19 (corresponding to Android 12) new release announcement changelog, April 2022, last accessed in Jan. 2026.

[30] A post on the Fairphone forum announcing the first Android 14-based build of LineageOS for Fairphone 3, last accessed in Dec. 2025.

while mainline projects like PostmarketOS and Mobian succeed in maintaining very few devices, but having a much more reliable maintenance system because mainlining and minimization of patching is here a primary focus. But all of these communities face the same frankenkernel and patching problems. As smartphones grow older and new hardware is released, the burden of kernel maintenance grows, while the number of users and maintainers for older devices shrinks, thus decreasing the possibility to sustainably maintain code.

Overall, it clearly appears from these observations, that the actors and factors driving obsolescence also limit the strategies implemented by alternative actors to combat it.

2.5 Discussion and perspectives

Our results show that the Android OS production pipeline generates frictions and lacks of incentives to support maintenance and upgrades in the ecosystem. We reflect now on the values and macro forces shaping this situation.

2.5.1 On the various types of software maintenance and their role in obsolescence

Towards the end of this study, we realized that the act of updating software is not or little questioned per se by the actors of the Android OS ecosystem. Our informants all assumed that updates are a necessity and that their absence is what causes obsolescence. This is what is defined as Lehman's law of continuing change [126]: "a program that is used undergoes continual change or becomes progressively less useful."

But do updates always respond to a need to mitigate obsolescence and make devices last longer in a useful and safe way ? We notice that each actor had its own way of approaching and envisioning updates and that all had different rhythms. It starts with the continuous development at the Linux kernel or derived OSes that follows, willingly or unwillingly, the endless changes in the hardware market. Then comes the yearly upgrades of Android that sometimes introduce new features, or new development behaviors that break retro-compatibility or make device updates more difficult. From phone vendors we observed few scattered updates before abrupt stops. Some alternative actors had a continuous maintenance process (PostmarketOS, Mobian), while others responded to breakdowns by attempting various forms of repairs or standardisation of practices (Fairphone, LineageOS, microG or UnifiedPush).

Both the diverging nature of updates as well as their different timing among actors of the Android ecosystem, play a central role in the fragmentation. Also, it remains unclear to what extent enforcing forms of backward compatibility could enable smartphones to remain functional without updates, as their broader software environment evolves.

Nevertheless, updates should at least be considered as ambiguous regarding the role that they play in maintenance. Updates are not necessarily only acts of maintenance and care for the devices and the ecosystem. They can and do also trigger incompatibilities and obsolescence. While updates are generally presented as a way to avoid obsolescence, updates from an actor become the obsolescence of another. Maintaining then becomes the obligation to cope with unwanted updates or the act of feeding the "monster" that the software to be maintained has become.

2.5.2 Values and choices in OS production and maintenance

From these different types of software maintenance we observe that "official" Android providers (Google, phone vendors) and alternative Android or non Android mobile providers define software quality in different ways, according to divergent values and interests. These different values then lead to different temporality in producing or maintaining code, and different software quality criteria.

Different values lead to different code quality criteria

Software support is a time and cost-relevant activity. Vendors and chipset manufacturers value putting new hardware products on the market at a frequent pace (every 6 months or every year) and have limited incentives to update their products.

The drivers, frankenkernels and software they produce are mostly oriented towards new devices. Once the device is on the market, they do not offer clear update policies for consumers, perform little updates and the end of support of old devices comes silently and quite often as quickly as two years after release. The recent European regulation (EU) 2023/1670 laying down ecodesign requirements for smartphones among other devices [31], in effect since June 2025, enforces OS security updates and upgrades for at least 5 years after the product has been released. How this directive will play out in reality, how smartphone vendors will implement it in practice, and how monitoring will be carried out, remains to be seen.

The Linux and alternative OS actors value open-source code, documented development practices, code longevity, and even hacking techniques, considering them legitimate. For them, updates and maintenance are continuous processes, following coding practices that foster collaboration and ease of updating. Alternative Android or mainline-Linux kernel communities place a greater emphasis on running on old, no longer supported architectures and having control on the software stack.

Google values its control over the ecosystem, as evidenced by recent changes aimed at gaining more control. In May 2025, the company announced[32] that Android OS would be subsequently developed internally in a fully private and closed-source way, before the code is pushed to its public branches once development is finished. This unilateral decision of closing development raised many concerns: for alternative OS actors maintenance becomes even more difficult as they can no longer perform maintenance as a continuous process[33].

Variations in the temporality of code production and maintenance

These diverging values affect the temporality of code production and maintenance practices among actors. Alternative actors are efficient in updating and maintaining, while developing functionalities or community collaboration methods are slow processes requiring much more effort. Vendors have slow maintenance and update rates, for short times, while often producing new frankenkernels for new devices.

[31] European Regulation (EU) 2023/1670 summary, last accessed in May 2025.

[32] Exclusive: Google will develop the Android OS fully in private, Android Authority, 26 May 2025, last accessed in Jan. 2026

[33] AOSP isn't dead, but Google just landed a huge blow to custom ROM developers, Android Authority, 12 June 2025, last accessed in Dec. 2025

Variations in code quality criteria

Code quality is also considered differently. Google and phone vendors will put greater efforts on user experience and standardisation. They will only green-light OS updates that fully pass functional and feature oriented quality tests at release time on every new device, while leaving aside maintenance work.

Alternative OS will favor code openness, privacy, security, sometimes accepting a lack of functionality (e.g. frequent issues with GPS or cameras on LineageOS, or with audio and battery gauge on PostmarketOS) while being fully transparent about this. Code quality in these communities will rather relate to the ability to facilitate collaboration and long term maintenance. The users are part of the development and maintenance process: they are expected to test and report bugs as developers fix them and push updates. Infrastructure and tools are provided for this functioning: forums where users and developers can interact and react on bugs and updates, as well as more technical tools for bug reporting or code contributions into the project repositories.

2.5.3 Power in the Android ecosystem

Google plays a central role in the Android Ecosystem. It owns the trademark and what can be called Android, it controls AOSP, its code and coding process, as well as the proprietary services and the core applications present on most Android phones. It also licenses the Android name and logo to manufacturers through the Android Compatibility Program (ACP)[34]. Android phone manufacturers that want to license Google's apps and services, are required by Google to enter an agreement called the Android Compatibility Commitment (ACC) [115, 116]. Previously called, almost ironically, the Anti-Fragmentation Agreement (AFA), it obliged vendors not to distribute any device based on an alternative Android OS alongside devices running on Google-Android. An antitrust legal case against Google by the European Commission (EC) in 2018[35], deemed it to be anti-competitive and to hinder the development of Android alternative OSes [112]. According to the EC conclusions, these tying practices consolidate Google's dominance and abuse of power in the Android ecosystem and exploit status quo-bias from users who are tied to Google Apps on everyday smartphone activities [127]. Similar antitrust infringement cases against Google have taken place in the US [114], in UK, in India[128], and more recently in Japan[36].

As a consequence, Google replaced AFA with ACC in Europe and vendors could then distribute alternative Android OSes on all of their devices. But they still have to follow the ACC guideline stating that only devices that signed the agreements can use and display the term *"Android"*, a registered trademark of Google[37]. These cases and changes in agreements between vendors and Google did change the situation a little in some regions. The vast majority of vendors continue to offer Google-equipped only Android phones. But some small phone vendors now offer, alongside the traditional Google-equipped Android phones, an alternative version with preinstalled de-Googled OSes: Fairphone since version 3 comes also with /e/OS, Shift phones come also with ShiftOS L or /e/OS, HIROH comes only with /e/OS, to name a few.

[34] See Android Brand guidelines and Compatibility Program, last accessed in Feb. 2025.
[35] European Commission case against Google, last accessed in Feb. 2025.
[36] On the Japan Fair Trade Commission's Google Decision: Some Early Reflections, Sangyun Lee, Kyoto University, April 2025, last accessed in Nov. 2025.
[37] Google starts blocking uncertified Android devices from logging in, Ron Amadeo, 2018 Ars Technica, last accessed in Jan. 2025.

According to Google [115], AFA and ACC are responses to the threat of incompatibility or fragmentation to Android. But our study suggests quite the opposite. Alternative Android OSes are concerned with maintaining updated devices, developing strategies of remediation to counter the lack of updates or maintenance largely responsible for the fragmentation problem. As for maintenance and updates, the definition of fragmentation is a matter of perspective, which varies depending on the point of view and values considered.

The same goes for power. SoC manufacturers, phone vendors and network providers have agency. The case of Fairphone demonstrates that vendors with limited resources can maintain Android builds twice as long as average, when they are willing to. As for Google, as repeatedly demonstrated by the various antitrust cases worldwide, it is able to exert power over vendors when it directly serves its interests. But when it comes to enforcing updates, Google's approach is much less coercive. Google ensures that the ecosystem provides a safe environment for vendors, by protecting their intellectual property and traditional production methods, and relies on their goodwill for maintenance. Meanwhile, Google's incentives to integrate its services and applications into the OS are much stronger.

2.5.4 On the many forms of openness

Google has always emphasized the open nature of Android. Google acquired Android Inc. together with its developers and founders, in July 2005, and in November 2007, announced[38] the first Android platform and OS version. The announcement also stated that Android development would be handled by the Open Handset Alliance[39], a consortium of many international phone vendors and network providers, led by Google. Here is how they were presented in 2007: *"the first truly open and comprehensive platform for mobile devices. It includes an operating system, user-interface and applications – all of the software to run a mobile phone, but without the proprietary obstacles that have hindered mobile innovation. [...] We hope to enable an open ecosystem for the mobile world by creating a standard, open mobile software platform. We think the result will ultimately be a better and faster pace for innovation that will give mobile customers unforeseen applications and capabilities. [...] Our goals must be independent of device or even platform*".

Our work shows that this openness is confusing in the Android ecosystem. Android is indeed based on open-source code developed in FLOSS communities, Google also produces some open-source code at the AOSP level, but inside and on top of them, lie layers of proprietary closed-source code, undocumented hardware or software functionalities, missing policies, and forced behaviors. The resulting OS is a "frankenware" system that the FLOSS communities have trouble understanding and maintaining. The narrative that is also pushed by Google is that of the security as a reason for OS upgrades and big changes in behaviour. But, as Korn and Wagenknecht argue, security maintenance is ambivalent [103], since some actors possess the privilege to take security decisions (industrial actors such as Google in this case) while others (alternative actors here) do not. Maintaining in the name of security then appears as another way of exercising power in the ecosystem, even if it means creating new maintenance issues that outweigh the original maintenance problem.

Recent developments around Android seem to confirm Google's confusion regarding the system's dual open and closed nature. In March 2025, Google announced a shift away from the open-source model for the Android operating system: source code would no longer be continuously published to

[38] Google Blog: Where's my Gphone?, last accessed in Oct. 2024.
[39] Wikipedia - Open Handset Alliance, last accessed in March 2024.

Android's public code repositories; instead, Google would develop it privately and release the source code in bulk, from time to time[40]. While industry actors continue to have privileged access to code, other actors do not. This endangers maintenance work by alternative open-source actors such as LineageOS or GrapheneOS, since sudden big chunks of code as explained previously are less easily updated, maintained or dealt with[41].

Furthermore, by announcing in September of 2025 a new mandatory *Application Verification Program*, requiring that app developers undergo a process of personal identity check and facial recognition verification, Google is throwing a wrench in the works for alternative open-source apps and app stores such as F-Droid—the main alternative open-source Android app store—and is gradually preventing them from continuing their operations. This made F-Droid and associated app developers launch the initiative *Keep Android Open* [42] signed by an important number of FOSS organizations and actors around the world, asking developers and legislators to act against this program as well as Google's growing dominance within the Android ecosystem.

FLOSS communities have formed their identity on the notion of code openness, reflecting on it not only as a technical value, but as a form of social development activity, and of political activism for building software as digital commons, considering them through the same lens as important public infrastructure, such as public roads, a metaphor used by Eghbal in her work on the making and maintenance of open source software [129]. But as FLOSS projects grow and become embedded in complex ecosystems like Android, involving Big Tech and other industry actors, they become part of what can be seen as what Ekbia and Nardi have called "heteromation" [130]: a set of practices of industrial actors to extract economic value from under-compensated or free labor in computer-mediated networks. Moreover, they analyse that this does not only conflict with values of openness towards digital commons in FOSS systems, this also affects deeply their ecosystem and the broader software ecosystem that we as users experience. As Geiger, Howard and Irani show, the activities and experiences of maintenance work change and are seriously challenged [82], as FOSS projects are embedded within these broader ecosystems.

2.6 Conclusion

In this chapter we studied maintenance practices around the Android operating system: a complex ecosystem that involves open source actors, as well as large and diverse industrial actors. Android OS is the result of their cumulative work, a complex software with billions of lines of code, which can run on highly diverse hardware and which is deployed on billions of devices. Despite this relative success, Android OS versions are short lived, Android smartphones are rarely updated, and compared to other devices, their lifespan is short.

Building on interviews with developers in the Android and other Linux-derived ecosystems, we show how Android OS differs from a traditional desktop OS: an Android OS build is unique to each device, which dramatically complicates the maintenance process. By mapping the Android ecosystem, the interaction between its actors, and the development flow of Android builds, we

[40] Google makes Android development private, will continue open source releases, Ryan Whitwam, ArsTechnica, March 26, 2025, last accessed in March 2026.

[41] AOSP isn't dead, but Google just landed a huge blow to custom ROM developers, Mishaal Rahman, Android Authority, June 12, 2025, last accessed in March 2026.

[42] KeepAndroidOpen initiative and open letter signed by 66 organizations from 21 countries, last accessed in April 2026.

show how the actors of this ecosystem have diverging and often conflicting update and software maintenance strategies.

SoC developers and phone vendors have little incentive to maintain the code of the Android kernels that they build for each device, let alone contribute back (upstream) to the Linux kernel on which they are based. This would imply following community conventions and practices that allow for code sharing, documentation, easy updates and better long term maintenance. This lack of contribution also shapes the way these actors code. They focus on their internal specific needs, producing code that increasingly differs from the open source code base, is little or not documented and contains several anti-patterns. This means that even if available, later opened or reverse-engineered, the code is particularly challenging to understand, reuse or maintain, denoting a lack of care and consideration for the community upon which it builds.

Key player in the ecosystem, we found that Google exercises its power in a selective manner, focusing on creating channels for selective updates, and enforcing its own proprietary services and applications while being reluctant to enforce upstreaming from SoC manufacturers and phone vendors. Long-term maintenance responsibility and work is transferred to free and open-source actors such as the Linux kernel community, alternative mobile OS systems such as LineageOS, Mobian, PostmarketOS, microG, UnifiedPush to cite a few, or to some phone vendors which emphasize longevity such as Fairphone. Because of lack of documentation, code accessibility, lock-ins and Google's applications and services, as well as its increasing dominance on the ecosystem, alternative actors have to put significant effort and develop circumvention strategies in order to maintain only some Android builds on specific devices for seven, sometimes up to ten years, or build open standards that could benefit to all.

Chapter 3

Maintenance strategies in Debian

Contents

3.1 Introduction

The Debian Operating System is one of the oldest and most widespread OS. Based on the Linux kernel, composed of free and open source software, Debian has a reputation of a stable and reliable OS among computer science professionals and the FOSS community. Debian serves as a basis for an increasing number of other derived distributions[1],—210 at the time of this writing—notably Ubuntu,

[1] According to Distrowatch, last accessed in March 2026.

Linux Mint, Tails, Kali Linux, Yunohost, Raspbian, to cite only a few of them. A timeline of the Debian family distribution tree, is shown below in Figure 3.1:

Debian's development began in 1993. It is nowadays used on a large number of servers worldwide[2]. Debian supports many different hardware devices: personal computers, servers, embedded devices, microcontrollers, super calculators etc. It also supports a wide variety of hardware architectures, called Debian ports[3], and in particular legacy architectures that are no longer supported by other operating systems. For example in Figure 3.2 we can see a social media post about the installation of Debian on an Intel 4004 CPU from 1971.

In online tech media, and in particular in general-interest articles offering advice on what lightweight OS could be installed on old devices, Debian or derived distributions are systematically cited[4]. For each of the supported architectures, Debian OS guarantees free of charge support for 5 years through the Debian Long Term Support (LTS) program. A paid Extended Long Term Support (eLTS) program exists for enterprises and professionals that can offer up to 10 years of support.

In *The Debian system book*, Krafft explores Debian, its historical foundation, and its organisation specificity [131]. Debian is not just an OS. Called the Debian Project, Debian is also a community of more than thousand active developers around the world as shown in Figure 3.3, contributors, individuals and member organizations. To cite the official Debian website project[5], the Debian community forms an *"all-volunteer"* and *"carefully organized structure"*[6], following the Debian Philosophy[7], a set of rules and guidelines that the community adheres to, formalized in the various following documents:

- the Debian Constitution, describes the organizational structure and the way the project makes formal decisions;
- the Debian Social Contract (DSC) and the Debian Free Software Guidelines (DFSG) describe the commitment to free software and to the free software community;
- the Diversity Statement, states how *"Debian welcomes and encourages everyone to participate, no matter how you identify yourself or how others perceive you"*;
- the Code of Conduct provides conduct rules for participants of online community tools;
- the Developer's Reference document, provides an overview of the recommended procedures and the available resources for Debian developers and maintainers;
- the Debian Policy, describes the policy requirements for the Debian distribution, the technical requirements that package must satisfy, etc.

Debian calls itself *"the universal system"*, and its community demonstrates a commitment to quality of maintenance, based on user needs, its open availability and wide spreading of the system.

[2]Debian and Ubuntu were used on 48,5% of all web-servers in 2023 according to w3cook.
[3]The Debian ports page, last accessed in March 2026
[4]See e.g. Best Linux distros for reviving an old PC, Tom's Hardware, April 2025; Top 10 operating systems for old PCS, Techies Nation, 2025 Edition. Last accessed in April 2026.
[5]See Debian people: Who we are and what we do, in the official Debian website, last accessed in March, 2026.
[6]See Debian's Organizational Structure, last accessed in March 2026.
[7]See The Debian Philosophy, last accessed on March, 17th 2026.

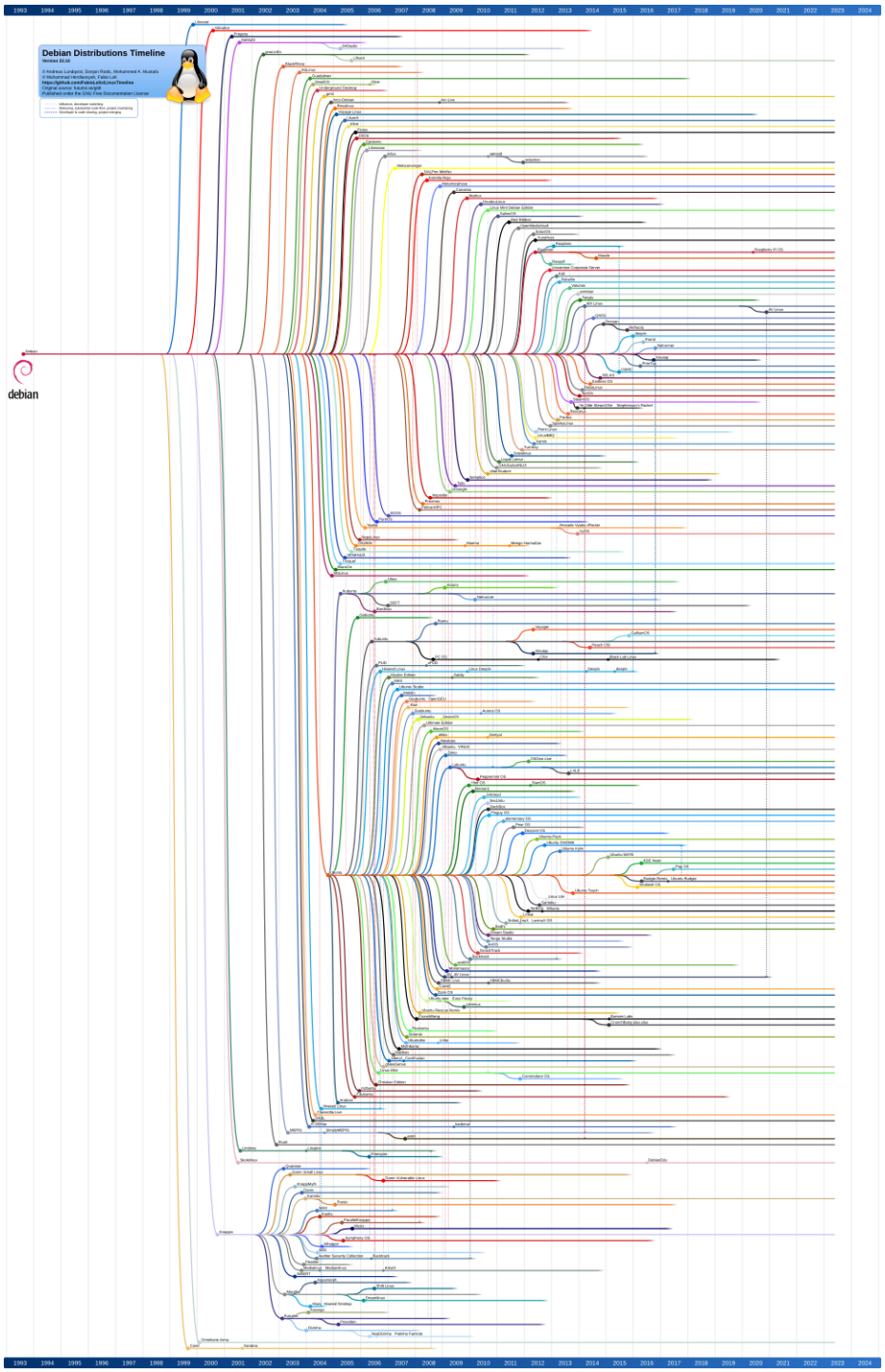


Figure 3.1: Debian family tree timeline. Andreas Lundqvist, Donjan Rodic. Modified by Michaeldsuarez. GFDL 1.3, via Wikimedia Commons.

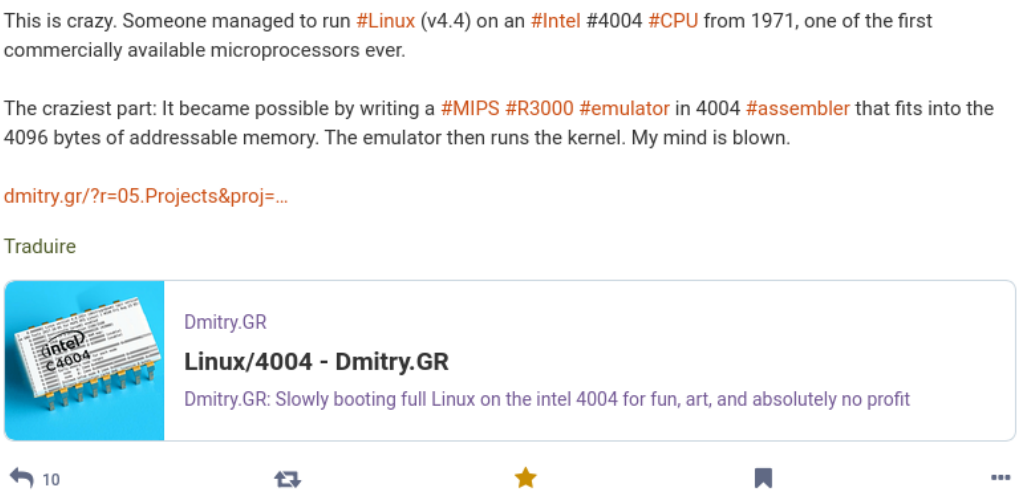


Figure 3.2: Screenshot of a Mastodon social network post pointing to Dmitry Grinberg's blog post describing the Debian install process on an INTEL 4004 CPU from 1971, last accessed in April 2026.

This is observed in the Debian Manifesto[8], a document written by Ian Murdock, Debian's creator in 1993, where he stated:

> "*Many distributions have started out as fairly good systems, but as time passes attention to maintaining the distribution becomes a secondary concern.[...] Debian may be used by many more individuals and organizations than is otherwise possible, the focus will be on providing a first-class product and not on profits or returns [...]. The Debian design process is open to ensure that the system is of the highest quality and that it reflects the needs of the user community. By involving others with a wide range of abilities and backgrounds, Debian is able to be developed in a modular fashion. Its components are of high quality because those with expertise in a certain area are given the opportunity to construct or maintain the individual components of Debian involving that area.*"

On the official website, in the *Reasons to choose Debian*[9], one can read that Debian offers *"stability"*, *"runs on numerous architectures or devices"* offering *"extensive Hardware Support"*, *"smooth upgrade processes"*, *"free of charge Long Term Support for 5 years"*, and software expertise as *"package maintainers do not only take care of the Debian packaging and incorporating new upstream versions. Often they're experts on the application itself and therefore contribute to upstream development directly"*.

In this study, I hypothesized that through time, Debian has built solid knowledge and practices in maintenance of software, update and upgrade issues, and software longevity that could help us learn more about software maintenance and remediation strategies to software obsolescence.

In order to study the socio-technical challenges associated with the development and maintenance of Debian, I carried out a multifaceted fieldwork within the Debian community. I participated in four of its gatherings from November 2023 to July 2025. During these gatherings I conducted semi-structured interviews with a diversity of community members: package developers and maintainers,

[8] the Debian Manifesto, debian.org, last accessed in April, 2026.
[9] Reasons to choose Debian, debian.org, last accessed in April 2026.

Figure 3.3: Generated map of Debian developers and locations as shown on map.debian.net as of March, 19th 2026. Map data from OpenStreetMap.

team leaders, community organizers, enterprises supporting and using Debian. I had numerous informal discussions with community members, participated in working sessions, attended multiple public talks and inner community discussions. I supplemented this community gatherings ethnography work with immersion into the online community websites and tools, together with an analysis of technical and research literature on Debian and maintenance related issues. Together with my research team, we performed a thematic analysis of the data gathered.

Building upon this corpus, I seek to answer the following questions:

- How is maintenance work in Debian structured and carried out? What are the different forms of maintenance conducted by the community?
- How is a Debian package developed and maintained in Debian? What is the Long Term Support (LTS) release and how does it tackle maintenance issues? What coding and social practices are involved in the processes of package and release maintenance?
- What inhibits and what favors maintenance in Debian?

After presenting related work on maintenance, open source software development and communities, and on Debian in particular, I detail my methodological approach. Our findings show that maintenance work in Debian is structured at different levels: the technical one, the social relations one, the infrastructure one and the organizational one. I detail this structure in section 3.4.1. Our results highlight the maintenance boosts (section 3.4.2) and barriers (section 3.4.3 that the Debian community encounters in its work.

Finally, drawing on the way that Debian deals with maintenance issues, I analyse and discuss the externalities that Debian provides to the larger software maintenance field, and the effects that Debian has on the broader software and hardware ecosystem (section 3.5). This is also an opportunity for me to discuss a broader research questions:

- How can maintenance strategies in Debian help remediate software obsolescence in general?

3.2 Related work on Debian

The Debian project and its community have been analysed in a quite large number of research studies from a variety of fields and points of view: from the technical and engineering point of view, focusing on its evolution as a system [132, 133], on its maintenance at the package level [133, 134], to the community organizational one [135, 136] and its evolution [137]. As an important FOSS ecosytem, with strong ethical values regarding the free software movement, Debian has also been the subject of multiple research from this legal, social and ethical perspective [135, 138, 139, 140].

3.2.1 An OS made of packages

Debian is the first Linux distribution to have structured its code in small units, called packages, that can be assembled together to form the whole Debian OS distribution. Packages are developed and usually maintained by one main person (called "maintainer"), sometimes a group of maintainers, without the need for each of them to be experts at developing a whole system, but rather by putting the pieces together in a bigger chunck that is the Debian system. It is at the level of the system that the discussions about how to organize it, what packages to include, as well as the work on dealing with inter-dependencies between the packages, and providing system-wide tools are located.

This modularity of Debian is one of the reasons that makes it a big community, as developers can contribute small amounts of code at the package levels. Ian Murdock, the founder of Debian, was himself quite impressed by seeing how this package-based system grew into a big community project very quickly [10].

But putting small pieces together in a coherent complex OS system is not an easy task, and not only from a technical point of view. In her work on recomposition [141] in 2003, author Rebecca E. Grinter emphasizes how, in building a system with pieces that have mutual dependencies, individuals engage in many forms of recomposition by communicating extensively, establishing roles, or informally engaging in it, while adapting the organization towards this goal. In Debian, the packaging management system that glues together the packages into the OS while resolving their inter-dependencies at installation and update times, has been extensively researched and improved by computer science studies [142, 143, 144, 145], in broad European Research projects (the Edos and Mancoosi projects) [146]. Release management in Debian has also been studied as a process, regarding system updates, security issues and management of inter-dependencies [147]. This research has helped advance technical work in system management more broadly, benefiting to other FOSS systems as well, and to maintenance work of software systems in general. In particular, they have helped to: better understand software interdependence in operating systems; avoid and manage conflicts during updates; build small and coherent operating systems by efficiently sharing and managing dependencies or avoiding code duplication that could unnecessarily bloat them.

3.2.2 An evolving project, restructuring itself around values and through challenges

As Debian is now a more than 30 years old ecosystem, it has evolved through time, in both its technical functional aspects as an OS, but also as a community. This is precisely what anthropologist

[10] The birth of Debian in the words of Ian Murdock himself, Glyn Moody, ArsTechnica, January 6th, 2016, last accessed in March 2026.

Gabriella E. Coleman has analysed in her book *Coding Freedom: The Ethics and Aesthetics of Hacking*, an ethnography work on FOSS developers, where she describes the social world of the Debian project [140] and how the community gathers and restructures around the OS.

Her paper called *Three Ethical Moments in Debian* [148] from 2005, has been of particular importance in my understanding of Debian, the ethical values that structure the project and its community. In both technical and social aspects, Coleman shows how Debian evolves in particular moments and through time. As Coleman testifies, Debian is a moving structure, adapting and restructuring itself around various values, technical, legal and social, that are constantly discussed and debated in the community.

This is how Coleman describes one of the three ethical moments, the *"ethical enculturation*" of members in the Debian community, through the process of becoming a new Debian maintainer:

> "*Although ethical enculturation is ongoing and distributed, the most pertinent instance of it in the Debian project is the New Maintainer Process—the procedure of mentorship and testing through which prospective developers apply for and gain membership in Debian. Fulfilling the mandates of the NMP is not a matter of a few days of paper shuffling. It can take months of hard work: a prospective developer has to find a sponsor and advocate, learn the complicated workings of Debian policy and technical infrastructure, successfully package a piece of software that satisfies a set of technical standards, and meet at least one other Debian developer in person for identity verification. This period of mentorship, pedagogy, and testing ensures that developers enter with a common denominator of technical, legal, and philosophical knowledge, and thus enter as trusted members of the collective.*"

This description by Coleman matches what I witnessed from my interviews and discussions with community members regarding the process of becoming Debian maintainers. From Coleman's description, it appears clearly that becoming a maintainer in Debian is not just a technical enrollment process. Debian development has a learning curve that requires commitment and time, not only because of technical considerations, but also because it is a social community with its own cultural and ethical values. Debian is open to every new contributor, but a great amount of work and effort is needed in order to properly contribute to code, to comprehend common rules, ethical and legal values, and follow processes that often embody important community principles.

Although the three ethical moments in Debian that Coleman describes, i.e. *"enculturation; the ethics of legal contrast and construction; punctuated crisis*" were observed many years ago, it was interesting, once I dived into my own ethnography work, fifteen years later, to acknowledge that they were still present in the community as Coleman had witnessed and described years before. Coleman's work added depth to my understanding of current issues and topics of interest in Debian, and were crucial to my findings in this research study.

For example, the fact that packages are now more often maintained in teams in Debian, and not by individual developers, is a change that came from what Coleman calls 'punctuated crisis'. Incorporate package maintenance into team dynamics as much as possible, and not isolate the package maintaining process at the individual level, was something that first appeared to me as a given process in Debian, as if it had certainly always been that way from the start, since it affects positively the resilience of packages. But in practice it was not there from the beginning, working in teams and maintaining in groups was an important shift in maintenance taken through time,

affecting the ways the developers work and the collective processes of maintaining. Realizing that work practices in Debian were not given realities, but came through discussions, sometimes conflicts and restructuring, was key to my understanding that in Debian maintenance is an ongoing collective process that changes through time, and not a recipe or set of rules that one has to follow, as the numerous documentation on package maintenance might, at first glance, lead one to mistakenly believe.

The 'punctuated crisis' that Coleman describes, highlight the moving nature of Debian and its re-structuring through time. But to my point of view, changes in Debian also happen in a more gradual way, as problems arise and the community identifies recurring patterns, discussing them on several occasions, and working together towards gradual, less abrupt, changes.

3.2.3 Not just a software system

In their anthropological work on FOSS communities, authors Coleman and Hill [149] have focused on how ethics of the Debian project in its community are reinforced through the sustained collaborative development of code, but also through discussions and decisions around licenses and policies. By describing what they call the *"ethical cultivation"* in Debian as observed in their ethnographic work within the community—the ways the community gathers around ethical values—the authors draw a model of what they call *"ethical volunteerism based on institutional independence, volunteer labor, and networks of trust"* that structures Debian.

Debian is not just a technical ecosystem, as the community of members—that are not only developers—and the collaboration between them play a key role in the functioning of the project as a whole. Debian acts as what Messerschmitt and Szyperski have described in their seminal definition of software ecosystems [150], *"collections of software products that have some given degree of symbiotic relationships"*. Authors Mens et al., in their work on evolving software ecosystems [151] describe *"bigger and more geographically distributed communities of developers"* that have made it possible to develop complex software systems by being geographically dispatched, using online tools over the Internet, especially in the open source development scene, and in Debian in particular.

Building upon this idea, in her work on reverse engineering software ecosystems [152], Lungu argues that *"a software system does not exist by itself"*, but as a part of a broader *"software ecosystem of projects that is developed in the context of an organization, a research group of an open-source community"*. The inter-dependencies inside a project are then not just seen as technical inter-dependencies of packages in a release, but as Hinchey argues [145]: *"developers contributing to this ecosystem may be involved in multiple projects and share implicit or explicit knowledge across these projects"*. Hence, the evolution of a project may be affected and affect connected projects also. This is particularly the case of Debian, as its relations with the broader FOSS community and the broader community of software ecosystems have an important role, as our findings in this research will also confirm.

Building upon these studies, Debian appears to be better analysed as a software ecosystem in the context of its surrounding ecosystems: the broader FOSS ecosystem, and further, the broader software development ecosystem. Community relations in Debian as well as relations to the broader connected ecosystems of software where Debian evolves, are to be seen as important factors in analysing Debian maintenance and its externalities.

3.3 Methods

My research in Debian was carried out on three complementary levels. First, I followed the Debian community in person during several gatherings from 2022 to 2025, and conducted community ethnography [153], by focusing on the maintenance work in Debian. These gatherings involved inner community work in the form of hacking camps during one or several week days, usually followed by two or three days of a formal conference gathering, with public conferences on Debian, gathering more members of the community, and being open to the large public. We also listened to selected online available conference talks regarding Debian taking place at Debian events worldwide.

Figure 3.4: Ethnography research on Debian. The Debian logo is licenced under Copyright © 1999 Software in the Public Interest, Inc., LGPL v3.

Second, during these gatherings I conducted 11 semi-structured interviews and had several informal discussions with various members of the community: Debian developers, package maintainers, event organizers, team managers, documentation writers, enterprises sponsoring Debian, etc. I also participated in working sessions.

Last but not least, I complemented my research by online information gathering reading technical documents or conferences on Debian, but also by following discussions on the different community online spaces.

My data corpus is represented on Figure 3.4.

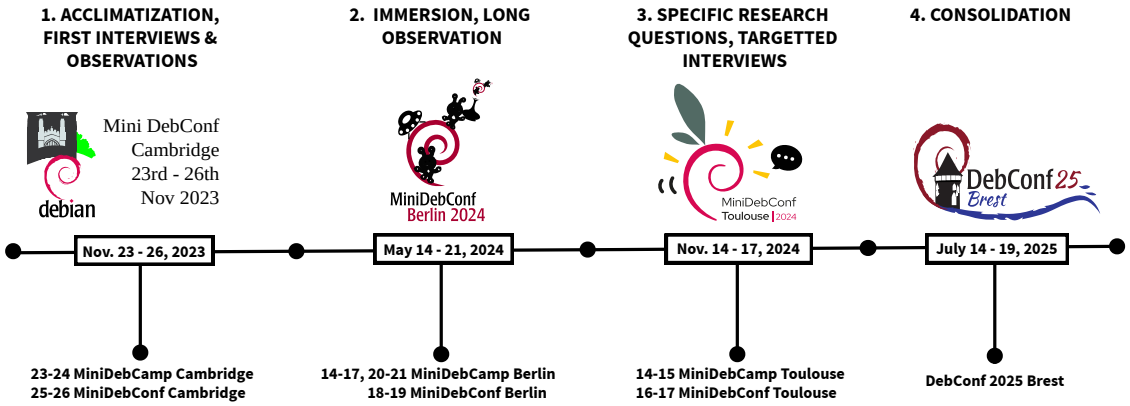


Figure 3.5: Timeline of attended Debian gatherings.

3.3.1 Community gatherings ethnography

I participated in person in four Debian community gatherings shown on Figure 3.5: the 2025 annual Debian conference in Brest and three European Debian conferences—miniDebConf Cambridge in November 2023, miniDebConf Berlin in May 2024 and miniDebConf Toulouse in November 2024. These multiple-day gatherings involved inner community work in the form of hacking camps during one or several week days (also called Debian hacking camps). These are usually followed by a more formal conference gathering, taking place during Fridays, Saturdays and Sundays, with public conferences and talks, gathering members of the community while being open to the large public. These gatherings also involved social moments during lunch times and dinner, organised tours or other informal events.

During these gatherings, I conducted formal interviews, informal ones and numerous discussions with community members. I also participated in working sessions, sometimes silently, sometimes being able to ask questions and once being able to participate in the work. The gatherings enabled me to meet with different actors of the Debian community: community members working in different Debian teams members, on both technical and non-technical issues, coming from various countries and sometime continents, speaking different languages and of various backgrounds. I met developers working on Debian, community members involved in organizing, enterprise employees working with Debian, Debian members working in enterprises that are involved in Debian, use Debian or are related to technical issues involving Debian. The gatherings allowed me to conduct interviews that were very important in understanding core issues in my research as well as informal discussions that helped me understand dynamics that did not surface in more formal communications. They were also an occasion to discuss my ongoing research work, and get recommendations and feedback on it from community members.

The gatherings played different roles in my research. The miniDebConf in Cambridge in November 2023, the first gathering, played the role of acclimatization. There I presented my PhD research as a lightening talk and welcomed people to come and talk to me about my research study. Many

community members came naturally to discuss with me about their work or giving me suggestions about people to interview. During this meeting I conducted two formal interviews (#1 and #2 in table 3.1) and attended several conference talks directly linked to my research.

The miniDebConf Berlin in May 2024, the longest field study of this research, served as a deep immersion and long observation gathering for me and my research. By the end of this gathering I felt as if I was a part of the Debian community, I felt welcomed by members and experienced their openness to discuss anything related to Debian. This is where I participated in working sessions and had numerous informal discussions and social interactions.

In the miniDebConf in Toulouse in November 2024, one year after my Debian field study had begun, I had a better idea of the main topics of interest that my research and field study was leaning towards, and had refined my research questions. This gathering helped me perform targetted interviews and consolidate my research. During the event I presented my research topic in a dedicated talk, my refined research questions, and my first results. I then welcomed the community to discuss them with me. I experienced once again a very welcoming atmosphere and interest in my research questions. I was very touched by one person that came after the conference and told me how happy they were that these topics were being discussed. I conducted several very important formal interviews during this gathering. I once again felt as if I was myself part of the Debian community.

The final Debian gathering, the Debian 2025 in Brest, was an occasion to consolidate my research. I conducted a final interview, had several discussions with older interviewees to ask for clarification or catch up on some issues, and attended numerous talks on Debian. As this was a much more bigger and international event, interactions with community members were less intimate but more varied.

3.3.2 Semi-structured interviews and informal discussions

I conducted 11 semi-structured interviews with informants from September 2023 to July 2025. Informal interviews or discussions also took place during the same period. Their informal character lies in the fact that they began naturally, were not scheduled, nor prepared, often not audio-recorded (by opposition to formal ones) but took the final form of long discussions on the research object very similar to the formal interviews. Personal notes were taken during or immediately after these informal interviews.

All of the interviews and discussions took place in person during conferences or gatherings. For each of them, we presented the purpose of the interview, the research work and explained how the information would be used. In an interview consent form, informants who agreed to talk to us could choose to appear anonymously or with their real name and professional activities. Furthermore, every participant received a copy of the writings (articles or thesis manuscript) that was submitted, with an invitation to review and discuss what they would consider as misrepresentations or errors.

I selected 17 important semi-structured interviews, informal discussions, and working sessions as shown in Table 3.1. These are the main data that I used for the thematic analysis presented below in 3.3.5.

3.3.3 Participant observation and working sessions

In miniDebConf Berlin in May 2024, the longest field study of this research, as well as in miniDebConf Toulouse in November 2024, a Debian camp took place during the first week days, before the public

#	Name	Team / Work subject	Function	Date	Event	Type	Duration
1	Emanuele Rocca	Debian & ARM porting	Debian maintainer & ARM developer	2023-11-25	miniDebConf Cambridge	Interview	1h
2	Federico Ceratto & Jochen Sprickerhof	Mobian team	Mobian developers	2024-05-16	miniDebConf Berlin	Interview	1h
3	Federico Ceratto	Debian community	Package maintainer, mentor & community organizer	2024-05-17	miniDebConf Berlin	Interview	1.5h
4	Johannes Schauer Marin Rodriguez	Debian & MNT Reform	Debian developer & MNT Reform port in Debian	2024-05-19	miniDebConf Berlin	Interview	1.5h
5	Ben Hutchings	Debian kernel team	Maintainer for the kernel team packages & the linux-firmware repository	2024-05-20	miniDebConf Berlin	Interview	1h
6	debacle	Debian & Ammonit	Debian maintainer, developer at Ammonit enterprise using Debian for wind and solar measurements	2024-05-19	miniDebConf Berlin	Interview, field notes	1h
7	Jochen Sprickerhof, Federico Ceratto & debacle	Debian developers, OrganicMaps package	Packaging OrganicMaps in Debian	2024-05-17	miniDebConf Berlin	Working session, field notes, online research	1h
8	Federico Ceratto	Debian developer, repackaging work session	Repackaging *anarchism*, a no longer maintained package	2024-05-17	miniDebConf Berlin	Working session, field notes, online research	2h
9	Rudolph Bott	Sipgate: VoIP and mobile telephony operator company	Working at sipgate, sponsoring the Debian event, using Debian internally and contributing	2024-05-19	miniDebConf Berlin	Interview, field notes	1h
10	Event organizers	miniDebConf Berlin org team	Community & event organizers	2024-05-19	miniDebConf Berlin	Informal discussion	1h
11	Mechtilde Stehmann	Debian maintainer & Debian women project	CoAuthor of the Debian Package Book : Building packages with Git-Buildpackage	2024-05-20	miniDebConf Berlin	Informal interview	1.5h
12	Arnaud Ferraris	Mobian & Linux on mobile teams	Team leader, maintainer, developer	2024-11-17	miniDebConf Toulouse	Interview	1.5h
13	Raphaël Herzog	Debian LTS team & Freexian	Team leader, maintainer, developer	2024-11-18	miniDebConf Toulouse	Interview	1.5h
14	Laure Herzog	Debion LTS team & Freexian	Developer for Kali Linux & Debian	2024-11-18	miniDebConf Toulouse	Interview	1h
15	Carles Pina i Estany	Debian Python team	Package maintainer	2024-11-17	miniDebConf Toulouse	Interviews	2h
16	Andreas Tille	Debian project leader	Debian project leader	2024-11-17	miniDebConf Toulouse	Informal discussion	0.5h
17	Helmut Grohne	Debian developer, working on the Debian /usr merge	Merging /bin & /usr/bin in Debian	2024-11-18	miniDebConf Toulouse	Informal discussion	0.5h

Table 3.1: List of Debian relevant interviews, discussions and working sessions.

conference gathering during the weekends. Every morning during the camps we held a stand-up meeting where we presented ourselves, our link to the Debian community, and our daily topic of work. This process helped me introduce myself and fit in naturally with the community.

The hacking camps are auto-organised spaces where people can work on Debian in groups or individually, can work remotely for their professional tasks, can participate in the social moments and informal coffee breaks, or discuss freely with community friends or colleagues. The hacking camps were very suitable for participant observation. Sometimes I would participate or listen to a discussion near me, related or not to my research topic, and other times I would work on my research via an interview, informal discussion or working session with community members. The Berlin hacking camp which lasted a whole week, helped me gather the most of participant observation notes and have varied interactions within the community. There, I participated as an observant in some working sessions between team members. I also participated actively in one of them, a session where we investigated what blocked the update of a precise Debian package.

3.3.4 Online community immersion and technical information gathering

To complement the ethnographic work, I followed online Debian discussions on the different community spaces such as mailing lists, forums, and collective IRC and Matrix chats.

I also analyzed historical and technical documents on the different Debian websites, serving dif-

ferent purposes, such as: technical development platform websites (salsa, git-packaging, qa debian), community news and information websites (Debian planet, Debian news, Bits from Debian) as well as community organization websites (the Debian wiki, the Debian gatherings websites.).

Almost all of the public conferences taking place in Debian gatherings are video or audio recorded and made available on the internet by the Debian video team. This allowed me to listen to selected online talks and explore certain topics in greater depth.

3.3.5 Thematic analysis process

The semi-structured interviews were audio-recorded with the consent of the interviewees. I automatically transcribed them to text first by using the Vosk offline open-source speech recognition toolkit [11], and then I performed a second, more thorough manual transcription. I kept a text journal of the most important quote excerpts from these transcripts, and what they triggered or highlighted. Many of them also appear in this article.

I performed thematic analysis [154, 155] on the data corpus shown in Table 3.1 composed of the full text transcriptions from the semi-structured interviews, and the field notes from the informal discussions and the working sessions. The coding of the data revealed themes and subthemes. These results were discussed on several occasions with my thesis director and team colleagues, through a process of iteration, coding review and cleanup, until we had the impression that no more iterations were needed. At the end of this process, the main results presented below had showed up as themes and subthemes. Thematic analysis was a novelty to me, so this also served as a learning process.

3.3.6 My position in the Debian community

As a long-time Debian user and as a FOSS developer and activist, I already was aware of certain aspects of the Debian OS and community.

In 2018, 5 years before this PhD began, I had volunteered to participate in the miniDebConf Marseille that took place in my city. On this occasion I had the chance to listen to conferences, participate in social events and get familiar with some community members. Thus, the Debian community and gatherings were not completely unknown to me when I began this PhD research.

Nevertheless, attending Debian events as a researcher was not the same position as that of the FOSS activist or Debian user and volunteer. In order to clarify this situation to the community as well as with myself, I clearly announced my research work in every gathering, on several occasions, as my main reason of attendance. The community was very welcoming. About halfway through my study, I presented my preliminary research questions and first results to the community, and we could discuss them collectively.

The last phase of this research is yet to come: first I am planning on sending my results to the interviewees and ask for their feedback on their participation; second I am planning to present my research and discuss it with the whole community, hopefully, during a future in person gathering, otherwise by online interaction.

[11] The Vosk Speech Recognition Toolkit repository

3.4 Results

Figure 3.6 shows an overview of the thematic analysis results that I will now detail. We characterize how the Debian community is structured around maintenance work and how this maintenance in Debian is located at technical and non technical levels (section 3.4.1). By focusing on the package development and maintenance practices during our interviews and field work, we could outline some software development practices and social practices particularly involved in package maintenance and longevity. By analysing the Long Term Support release model we were able to investigate financing and organisational strategies that Debian has implemented in order to offer a 5 year long support on the releases.

Our findings highlight what technical, social and organisational factors help boost maintenance in Debian (3.4.2), and what are the main obstacles and challenges that the community deals with in tackling maintenance of the Debian system, but also of its own community (3.4.3). Further reflecting on these results, we highlight in section 3.5 how this maintenance in Debian extends outside of Debian itself, and has positive impact on broader software and hardware maintenance, preventing and addressing their obsolescence or premature end of life.

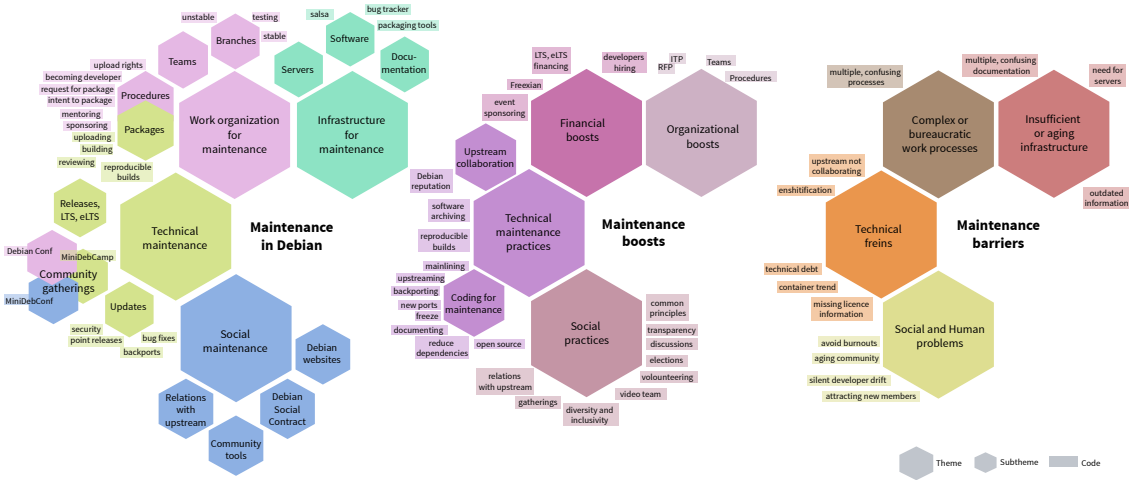


Figure 3.6: Maintenance in Debian as identified through thematic analysis.

3.4.1 Maintenance work in Debian

Thematic analysis of our data helped us identify how the technical and social organisation in Debian is carried out for and in support of maintenance. This maintenance is located at different levels in Debian, following the main subthemes in Figure 3.7:

- at a technical level,
- at the working processes, organisational level,
- at an infrastructure level,
- at a social level.

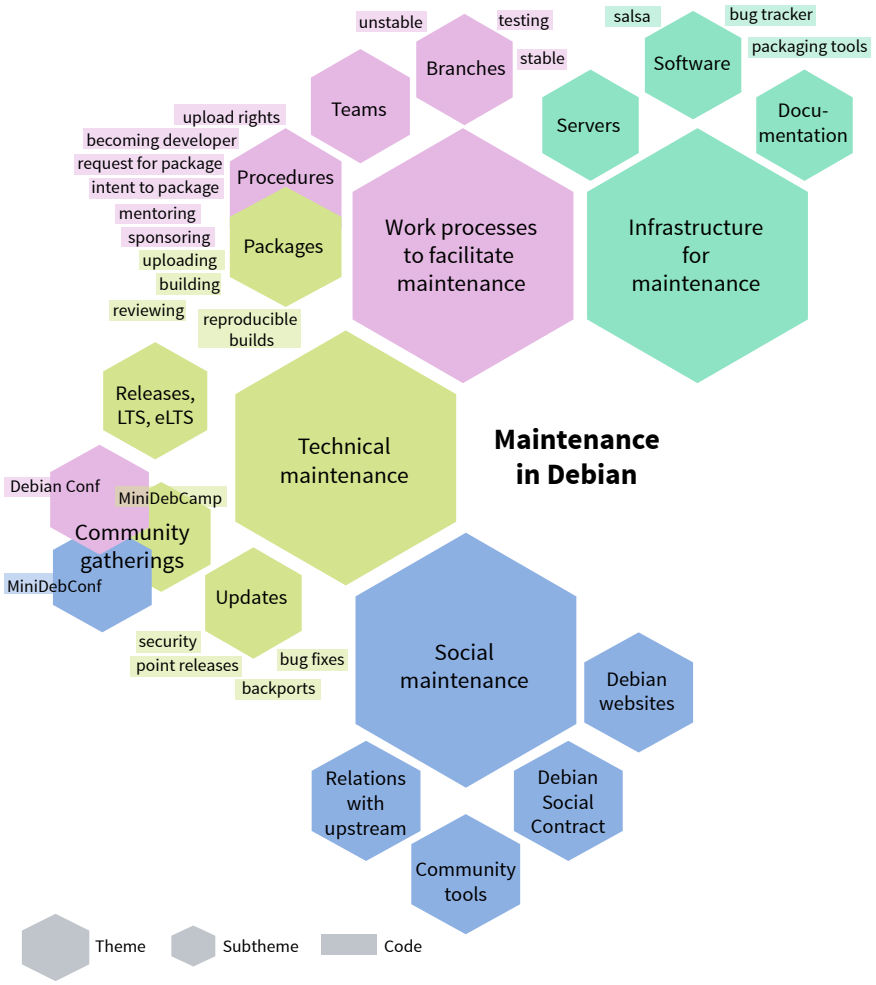


Figure 3.7: Maintenance location in Debian as identified through thematic analysis.

Code maintenance in Debian

Code in Debian is organized in units of packages to be developed and maintained. A Debian release is then a set of selected packages that are carefully put together so that they can form an operating system, on top of which users can add additional application packages. The packages that form the Debian system follow requirements stated by the Debian Policy related to technical functioning and inter-dependencies between them, while their software licences have to comply with the Debian Social Contract (DSC) and the Debian Free Software Guidelines (DFSG).

Packages are dynamic entities that follow a lifecycle of development and maintenance. New packages are requested, developed and then introduced into Debian. They are subsequently maintained on an ongoing basis through security updates, bug fixes, or code changes needed from update of packages on which they depend, or inherited from the source code from which these packages are derived.

Maintenance plays an important role in the life of a package. Packages may be declared abandoned and consequently removed from Debian releases. They can be awaiting re-inclusion, when their maintenance stops, which may occur due to a lack of Debian maintainer for this package, or

because the source code from which the package originated, the upstream code, is no longer maintained. A life and death evolutionary study on Debian packages has been undertaken by Raymond Nguyen and Ric Holt [156]. In their work they draw a biography of Debian packages by examining four essential attributes: their age, their bugs, their maintainers and their popularity.

During the study I witnessed the creation of the OrganicMaps package, a map open-source application available for Android. This first came as a discussion between participants n°7 in 3.1. One of them was a user of the app, while the two other participants were members of the Mobian team, the team porting Debian to mobile devices. The latter were also interested, and had the best knowledge for adapting phone applications into Debian packages. Little by little during the hack camp in Berlin, they began to build the package, spot the changes needed to be made on the upstream project, the OrganicMaps app, and initiated contact with the person developing the application (the upstream developer) in order to ask for the changes to be made upstream.

Another working session during the hacking camp consisted in maintaining the *anarchism* package—a package offering a way of locally downloading and consulting in various formats the online *An Anarchist FAQ* book. The package was showing errors when built for the new Debian coming release called *trixie*. During the working session we identified problems depending on upstream developers, and began searching their contact in the Debian package information tool. We realized that similar attempts had been previously made to contact upstream developers, without success: the package upstream developers were not responding. Maintaining it inside of Debian was considered during the session, by evaluating the work needed. After carefully checking existing bug issues filed on the package, we filed a *wishlist* bug in the Debian bug tracking system, as a request for the package to be updated by its team of Debian maintainers. As I noticed two years after, in February 2026, the Debian package tracker tool for anarchism [12] shows that upstream maintenance is still not being met, but that Debian developers have upgraded the package, fixed the blocking errors and have included it in the latest Debian stable release, *trixie*.

Packages can depend on each other. Part of the maintenance process of a package in Debian is resolving package dependencies and maintaining them, as one package change can involve updating and maintaining dependent packages. This Debian package inter-dependency system itself has been the subject of significant research and improvement [142, 143], and is a key component of the Debian system. The package management system is therefore an important component of the Debian system, that need to be maintained and improved. Several research studies have done so [144, 145] and several research European projects, e.g. the Mancoosi project [13] have been dedicated to the issues of analysing and improving FOSS distributions, their updates, upgrades and packaging systems, in particular in Debian.

In Debian, there are several tools to help automatically build a package (Packaging Tools), test and check for errors (the Lintian tool), review their state (the Debian tracker), repositories to host their source code and follow the changes (Salsa), or to report and track bugs (the Debian Bug Tracking system) [14], to cite only a few of them. These tools help package maintainers and developers in their maintenance work, as well as others to better know and follow the current state, as well as the development and maintenance processes in Debian. These tools have evolved during time, and are also the subject of technical maintenance work, as we witnessed in several conference talks and

[12] See Debian tracker data for anarchism package.
[13] The Mancoosi project, last accessed in March, 2026.
[14] See the Packaging Tools wiki page, Lintian, salsa, The Debian Bug Tracking System.

community discussions dedicated to them.

Work processes, teams, roles: Debian's inner organisation towards maintenance

The Debian organization appears to be structuring itself for and around maintenance. Organized in approximately 200 teams [15], working on various. There is a Kernel team, a Security one, the Mentors team, the Community one, the Women team, the Data Protection, the DebConf, the Backports, the Mailing Lists team, etc. Teams have their own tools—website, documentation, repository project—and their own communication systems—emails, website, mailing list, etc. At least one official guide for helping create, manage and announce teams exists. Team members can have different team roles, depending on the team: team leaders, project coordinators, subproject etc. The inner organisation of a team seems to be decided among team members in a self-managed, self-determined way. Working in teams seams more sustainable than individual isolated work: if a volunteer is not available, work can be taken over by other team members.

Coming back to the packaging management work, dedicated roles have been imagined by the community: package maintainers, package developers, package sponsors or package mentors. A package maintainer, also called Debian maintainer, is a relatively new member of the community that takes over the work of packaging and maintaining one or more package. The Debian maintainer can maintain a package without having to go through the process of becoming a Debian developer (DD), but has limited rights in Debian. A Debian Developer has further packaging rights, in particular the one to sign and upload the package to the Debian official packaging system, so that they can, after dedicated processes, become part of the Debian OS releases. A DD is also a Debian community member that has the right to vote in Debian elections. Becoming a DD follows a specific documented process in Debian. Another dedicated process is available to Debian developers allowing them to gain superior upload rights for any package, not only the ones they are maintaining, into the Debian official repositories.

It is interesting to note how these hierarchy and processes to become maintainer, developer or gain full upload rights, are codified. Carles Pina i Estany (participant n° 15) a junior package developer of less than a year, described in an interview his process of becoming a DD:

> "*You can apply for being a developer and then you follow a process where other Debian members ask to you a lot of questions: it is about the Debian philosophy, licenses, technical aspects, how to collaborate in Debian, how the voting and the elections work, etc. The idea is to see if you understand the philosophy and if you are ready to work in a certain way that is expected from the community. All questions are public, but you respond privately, by email. [...] If you want to have full rights in uploading packages, you then have another set of questions where they ask more technical details on how to package things. These questions are public as well.*"

These processes are taken over by a Debian role called application manager (AM). Carles Pina i Estany continues:

> "*They can ask questions which are not always in the template. When you answer, your application manager asks follow-up questions. For example they say: 'explain the differences between license A and B', and then you explain by mail. The questions are public,*

[15] See the Debian teams wiki page, last accessed in Feb. 2026.

> *the mails are private. The AM has access to this, and I think all other AMs do: it is a kind of front-desk team. Then there is another step where someone else validates this conversation exchange. It is an interesting exchange and it can get quite long, with long emails.*"

From a technical point of view, becoming a package maintainer and developer is a long process, but where you are not left alone in Debian. Dedicated roles called Debian package sponsor, also called Debian mentors help new maintainers to package. According to the official Debian documentation: *"Sponsorship means that a Debian Developer uploads the package on behalf of the actual maintainer. The Debian Developer will also check the package for technical correctness and help the maintainer to improve the package if necessary. Therefore the sponsor is sometimes also called a mentor."* [16] Carles Pina i Estany describes the mentoring that he benefited from:

> "*Before becoming a DD I did not have the right to upload my packages. This was the role of the sponsor: you send a package to someone to review—[Name Surname] in my case has been sponsoring many of my packages—he gets them, checks the quality, tells me things that maybe I missed or that he would change or improve, or he thinks we should do differently, I do them and then when he's happy and I am happy he pushes them to Debian.*"

Carles Pina y Estany told us that in becoming a DD, experience is key *"as every package is different and comes with its own issues, and when you build 10 packages or more, you begin to better know how to do this work*". But without the mentors and team help, in his case the Debian Python team, he is not sure if he would have had the motivation or the possibility to continue alone.

We took here the example of the packages, a technical issue, but Debian developers are not only technicians. Carles Pina y Estany explains that *"[one] can be part of many different teams, be a Debian developer and not do any packaging: in the publicity team, in servers team, translation team*". Other roles were also present in specific teams, specific to the team's work, as we will further describe below, when describing the social maintenance in Debian.

Teams, processes and roles seem to have an important role in defining and maintaining the structure of the Debian community: roles helping to welcome new members, to include them in community work, to help them gain further rights in it.

Infrastructure maintenance in Debian

There is a solid and diverse hardware and software infrastructure helping Debian exist as a system, and as a community. The physical infrastructure is made of more then 150 servers that are maintained by the Debian system administration team and that are necessary to build Debian packages, and host the Debian technical inner working tools: Salsa, the bug tracker, Lintian, security.debian.org, ftpmaster.debian.org, QA, and many more. Another important hardware infrastructure are the Debian servers that act as web mirrors available at different locations worldwide in order for all Debian users to download and install the Debian system, but also, to update and upgrade them. This mirroring system is dynamically constructed so that it is the nearest server which gives a user the packages to download. These mirrors are not only Debian owned: they

[16] Debian Mentors web page.

are sometimes hosted and provided to Debian by universities, companies or cloud providers. This hardware infrastructure also includes the numerous Debian websites and wikis documenting Debian.

During Debian gatherings, several conference talks focused on this physical infrastructure, the needs to maintain it, to update to newer servers, or the need for more of them. The question on how to get them at a free or reasonable financial cost was also raised and several discussions on why and how to deal with the problems were exposed.

The software infrastructure in Debian consists of all the software tools that the community has developed and maintains for its inner needs: the bug tracking system, Lintian, the Quality Assurance tools, to cite only a few of them. These are tools that are sometimes fully developed internally, or open source tools that have been adapted to Debian needs before being put into service. Before Debian has very specific guidelines and internal processes, this is often the case with external tools. The development of these software infrastructure tools was also often discussed in talks. The development cycle seems to follow numerous phases of development, testing, discussion, improvement, before being officially adopted and deployed in production.

The numerous Debian websites serve as entry points to the Debian system, offering information and installation information, but also to the community, offering social ways to meet and coordinate work. They also serve an important purpose: documentation. Indeed, the Debian websites and wikis seem to document everything: how to install or update the system as a user, how to build and maintain a package as a Debian maintainer, how to build a team, to report bugs, etc. These tools are maintained by dedicated teams and members of the community. For example the Publicity team, maintains the official news, announcements websites and social accounts (such as Bits from Debian, the official Debian blog). Each team maintains its own documentation regarding its team work. All of these websites and documents are translated in a large amount of languages by volunteer members on a per case basis.

Mechtilde Stehmann, participant n° 11, described her work in writing and maintaining the Debian Package Book [17], a book documenting each step of the packaging work, and showing by examples some common difficulties and ways to handle them. In the long process of writing, she had to extensively discuss things with other fellow Debian developers involved in packaging, and consider the diversity of difficulties that a person trying to package could run into. She told us how difficult this part was, and how, even if automatic packaging tools and processes exist, every package is unique, and every Debian developer has built its own techniques in making packaging easier. She told us how documentation is not just about writing a simple technical recipe. In a community like Debian it is also about co-development, discussing issues, and building together.

Another example is that of the Quality Assurance (QA) in Debian, and its dedicated team that develops and maintains a set of tools which *"aim is to improve the Debian system as a whole, not only a specific set of packages*" as written on the team's web page. These tools range from the Lintian tool that builds and checks packages, to the Debian Package Tracker and the Developer's Packages Overview that inform about packages and developers, deal with mass filing of bugs, with emergency maintenance of important orphaned packages. But, as the team states on its web page: *"[Quality Assurance] also serves as a central place for discussion about distribution-wide problems and improvements regarding quality*" [18]. What these examples show here is that infrastructure work participates in the collaboration and co-construction of Debian. They also show the articulation

[17] The Debian Package book, last accessed in March 2026.
[18] See Quality Assurance in Debian.

role that infrastructure plays in the Debian community: serving as a bridge between external needs regarding the Debian system and inner Debian community needs, between community members and external ones. This is what has been described as *infrastructuring work*: an ongoing collaborative socio-technical process towards the long-term, the longevity of the system [157].

These examples show how the infrastructuring work in Debian plays a collaborative socio-technical role in the maintenance of the Debian system.

Social maintenance in Debian

Social relations in Debian are crucial. Most of the essential processes in Debian are social relationships between members of the community: mentoring Debian maintainers in building a package, learning how to develop and maintain in a community with package dependencies, technical and ethical requirements, filing an issue is communicating it to the broader community, etc.

These social relations extend outside of Debian. The process of building a package for an external software, the relationship with the original author of the software, the upstream author, is crucial. Free open-source systems give the right and the possibility, as source code is available, to freely take it and modify it, without having to ask for permission to the authors, the upstream developers. From a maintenance point of view, in Debian this is not a good idea, as upstream developers are the best fitted to maintain and develop their own software. The Debian packaging work consists in taking into account and integrating in the workflow the upstream maintenance work: bug fixes, security fixes, new features, upgrades to new systems etc, should be ideally made on the upstream project. This relation is two-sided: the upstream project developers and Debian package maintainers, can help each other by maintaining their specific parts: the Debian developer can report bugs, licence issues, or even code patches; the upstream developers can help Debian packaging with good licensing requirements and good coding practices (documentation, available code, bug report systems etc). A good relationship between the two actors is key to easy maintenance of the package. As Carles Pina i Estany, who maintains several python packages in Debian, told us:

> "*I talked to all of the upstream developers for the packages that I maintain in Debian, because during the packaging I discovered bugs and so I also helped the upstream code to correct them. The majority were very happy to have their package in Debian and to collaborate. All of them responded, but one of them, not active anymore. So actually this is the one package that worries me. At some point I even went to a Python conference to explain how to make your package easy to be maintained by Debian, because an important thing in Debian is to avoid dependencies that are not really needed. For example, one of the packages I had, uses something that I am not sure it needs, maybe it could be avoided.*"

The importance of the relationship with upstream appeared in many interviews and interactions. It is interesting to note that, as we witnessed during our working session on the updating of the anarchism package, this social interaction is also codified in the package maintenance processes: the upstream is regularly checked for updates by automatic tools sending automatic mails to Debian package maintainers when they detect an update. Errors in the building process are automatically reported via mail to upstream maintainers. The Debian Package Tracker tracks all interactions or events regarding a package. The tool is latter automatically linked to the QA for the package, showing bug fixes and corresponding emails exchanged between all parties working in maintaining

the package. Once again, this demonstrates the role of infrastructuring work in maintenance, and how social interactions intertwine with technical work in the maintaining processes in the Debian.

The social component in Debian is even more visible in non technical work. A Debian developer is not only a technician that maintains code and packages: members of the Community, Debian Women, Diversity, Accessibility teams, support inner community members but also Debian users, and help making the community as well as the Debian system welcoming and accessible. The Debian Diversity team puts focus in welcoming everyone by preventing discrimination, establishing a Diversity Statement and making sure it is followed, or providing help and a point of contact for LGBTIQA+ people [19]. The Accessibility team, one of the oldest teams I remember in Debian, creates tools in order for the Debian system to meet the requirements of users with disabilities, while also playing a role in the Diversity, Community and Publicity teams in making the community more accessible.

At the Debian release level, authors Di Cosmo et al., in their paper on package upgrades in FOSS distributions and in Debian in particular [144] further observe that: *"Distribution maintainers act as intermediaries between "upstream" software authors and users, by encapsulating software components within abstractions called packages*". This intertwining between technical and social aspects of Debian, and how they help and build on each other, is also visible in the Debian Popularity Contest, an initiative, but also a package that once installed gathers statistics from users on the Debian packages that they use, in order for Debian to better put maintenance efforts is what users use the most. In their work on modeling software inter-dependencies by using Debian, German et al. highlight how in FOSS systems it is common to build new software by taking advantage of a rich and complex environment of other FOSS available programs and libraries [142]. They describe the Debian Popularity Contest by saying:

> "*The Debian Popularity Contest is an attempt to map the usage of Debian packages. Its main goal is to know what software packages are actually installed and used. This information is used in order to determine the order in which packages are put on [installation] CDs [...] and it is also used during quality assurance activities as a criteria on which packages to focus.*"

3.4.2 Maintenance boosts in Debian

Maintenance boosts and barriers are also located at different levels shown in Figure 3.8: at the technical maintenance practices level, at the community social practices one, at the infrastructuring one and at a new level, the financial one. We now detail each of them.

Technical maintenance boosts in Debian

Upstreaming code was mentioned regularly in our interviews or observations as being a fundamental principle in developing code in Debian and more widely in FOSS, when building upon an existing software. All of our interviewers explained how fixing the bugs, removing unnecessary dependencies, fixing licensing issues during the Debian packaging process, were better done as updates in the upstream code by upstream developers themselves. If not, on the next update of the upstream code, when the Debian package would also need updating, the same problems would appear again

[19] See Diversity team creation announcement, 29th of June, 2019.

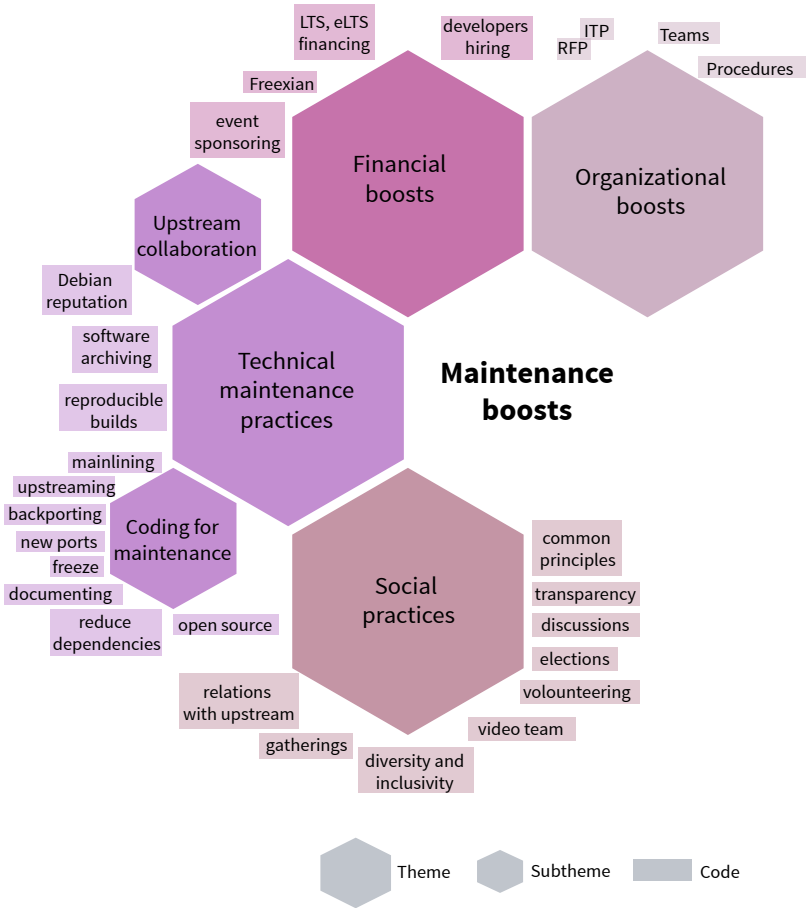


Figure 3.8: Maintenance boost factors in Debian, as identified through thematic analysis.

and maintenance work would once again be needed, duplicated. Doing things upstream resolves the problems at the root, maintenance is necessary only once, and maintenance work does not accumulate while the software upstream keeps being updated.

At the kernel level, the Debian kernel team puts its focus on mainlining the Debian kernel to the Linux one, i.e. closely following every little update or upgrade of the kernel in a continuous pace (sometimes only hours or few days separate a Linux kernel release from a Debian kernel one). By doing so, as the Android OS example shows, Debian stays updated with all security issues, upgraded to the latest Linux kernel, and maintenance work is an ongoing process of updating little chunks of code at a regular pace.

Let us now focus on some other tools helping Debian maintenance that we encountered during our research. Reproducible builds, mentioned several times, is a relatively new project in Debian. Its aim is to make sure that when rebuilt in the same conditions, a source code results to identical binary code builds. The tool also shows whether generated binaries correspond to the original source code or not. Thus, reproducible builds guarantee the security of the build environment and of the source code itself. As research work have shown [158], reproducible builds helps maintenance by increasing the integrity of software supply chains. As Carles Pina i Estany (participant 15) said when talking about *simplemonitor*, one of the packages he maintains:

> "*The standard Debian package pipeline tests for reproducible builds. It also helps finding bugs. Simplemonitor was not reproducible, which means that the code was corrupted, either there was a bug or a line of code that should not have been there. In that specific case, depending on the time zone that you were building the package, the generated binary was not the same, and also some unit test was not passing. It was because the simplemonitor developer is in the UK and never tested it with an environment in minus twelve hours timezone like I did. I reported this bug upstream and it was fixed.*"

Similarly, Software Heritage, *"an initiative of building a long-term universal archive for software source code, capable of storing source code files and directories, together with their full development histories*"[159, 160, 161] was mentioned as a tool to help with code licensing issues. Federico Ceratto, participant 8, a senior package developer and community member described to us the importance of licence checking in Debian when building a package, as Debian has a strict licensing policy (described in the Debian Free Software Guideline). He explained how this was a very tedious work, not only because of the legal aspects, most of the time because licensing information is missing upstream. When a package has several library or software dependencies, which is usually the case, every dependency has to be checked for its licence and its compliance to DFSG.

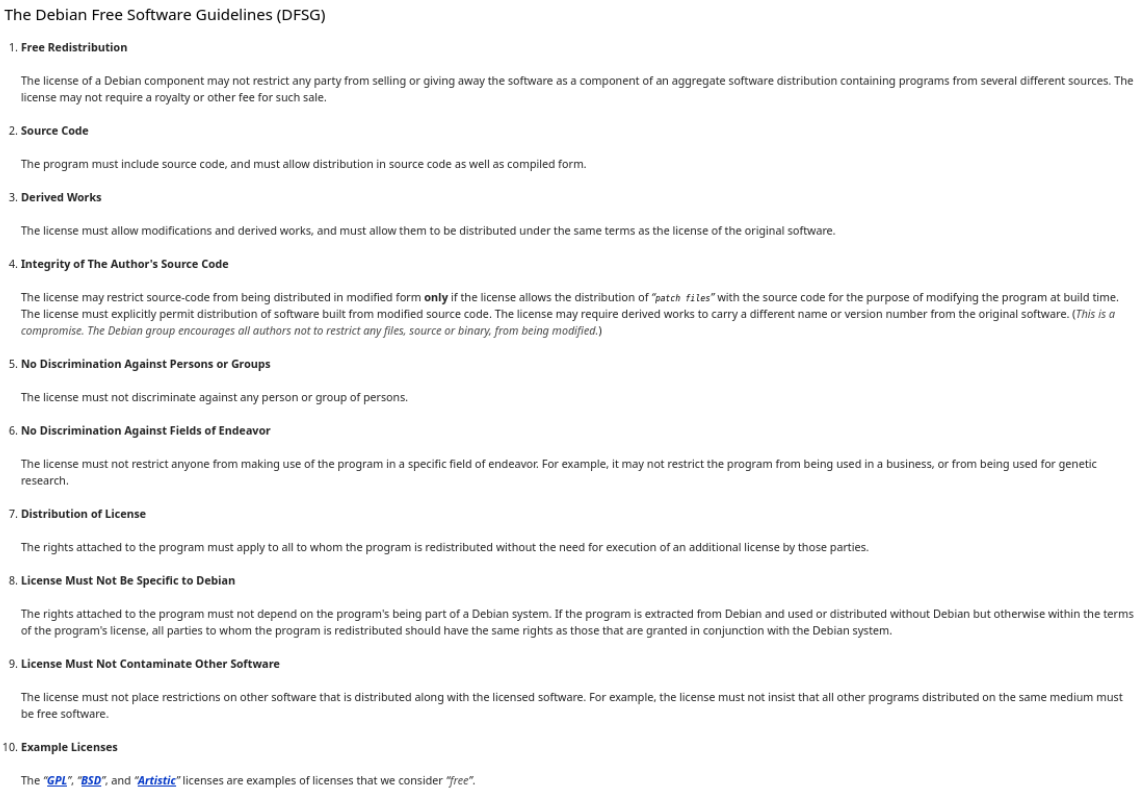
> "*While packaging you might have to do a little bit of work that you are not supposed to do. Especially for checking licences. For example once I had to build a package using upstream logos which were not GPL, but the project was. I looked to find the same icons and their licence on the web but I could not. I had to redesign the exact same icons with a GPL library, while I am not an icon designer at all. Today I was talking to [pseudonym of another Debian member] on their work on Software Heritage, where they are building this massive database archive of all free software contents in the world. In my use case it is exactly what this can solve: if I have an image I can look up on this database to see if there is the same content somewhere else. Because in this database the content is addressed by the content of the file and not just the file name, if I find an image called logo.png (or source code) and somebody else has the same image but called cat.png, (or same source code content) it will tell me if it is the same. This could have really helped me in finding the licence of the logos.*"

While the important role of archives in preserving history, cultural heritage, fostering research and studies, is quite established—in particular in humanities, arts or social sciences—this example shows their interest in developing and maintaining software.

Ceratto explained how the Debian Free Software Guidelines (DFSG) are not just a policy requirement, but also a helpful tool in assessing license compatibility in a package:

> "*Occasionally you find a weird license that you actually have to look at. The DFSG helps you assess and understand if a licence is resistant or not. It's like exercises to help you assess if you can use this software freely. For example one of the exercises is 'if you are on a desert island disconnected from the rest of the world, can you use the licence?' Because some of them say you can use freely but you have to login here or you have to email there, this will not work, you have to be able to use the software even if you are not connected to the Internet on a desert island.*"

It is interesting to note here how the DFSG serve multiple purposes. If we take a look at the principles (see figure 3.9) they describe the founding ethical and legal principles of Debian as a free and open-source project. From what Ceratto described, they also act as a technical and legal tool for verifying the nature of a license while developing in Debian. And finally, from the exercise highlighted here, they also act as a tool to reflect on the world and how Debian fits into it. As such they do not only help with technical maintenance, but they are part of the ethical enculturation of community members that Coleman describes, within Debian and also outside of it.



The Debian Free Software Guidelines (DFSG)

1. **Free Redistribution**

   The license of a Debian component may not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license may not require a royalty or other fee for such sale.

2. **Source Code**

   The program must include source code, and must allow distribution in source code as well as compiled form.

3. **Derived Works**

   The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

4. **Integrity of The Author's Source Code**

   The license may restrict source-code from being distributed in modified form **only** if the license allows the distribution of *"patch files"* with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software. (*This is a compromise. The Debian group encourages all authors not to restrict any files, source or binary, from being modified.*)

5. **No Discrimination Against Persons or Groups**

   The license must not discriminate against any person or group of persons.

6. **No Discrimination Against Fields of Endeavor**

   The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

7. **Distribution of License**

   The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

8. **License Must Not Be Specific to Debian**

   The rights attached to the program must not depend on the program's being part of a Debian system. If the program is extracted from Debian and used or distributed without Debian but otherwise within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the Debian system.

9. **License Must Not Contaminate Other Software**

   The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be free software.

10. **Example Licenses**

    The *"GPL"*, *"BSD"*, and *"Artistic"* licenses are examples of licenses that we consider *"free"*.

Figure 3.9: The Debian Free Software Guidelines, as of April 2026.

Organizational maintenance boosts in Debian

We previously talked about the organizational maintenance practices of Debian, teams, roles and processes and how they structure maintenance work in Debian. These organizational practices appeared often in our observations as boosts to maintenance. For example Carles Pina i Estany (participant 15) explains, when talking about becoming a member of the Debian community:

> "*I was just a Debian user, not a developer, and I needed some python software to be in the system. I looked at how it was supposed to work, and I saw that I had to open a request for packaging (RFP) bug, and that is what I did. No one did the package. So I decided I would do it. This was a year and a half after my RFP. I then changed the bug from RFP to an Intent To Package (ITP) bug and said in it I will do the packaging, if anyone wants to do it with me get in touch, we can do it together.*"

These structures offer the community ways of being aware of the work that has been done and is left to be done, ways of easily integrating the community, of joining a team or a project that is already underway, of peer learning or of collaboration between teams. The maintenance process in Debian appears as a community-driven approach, boosted by the community tools and inner organization, pushing towards building things collectively, as opposed to putting the main focus at the individual level.

Social maintenance boosts in Debian

Debian's reputation was mentioned by several informants as an important factor that fosters upstream collaboration. Since Debian has a strong reputation of technical excellence, when a Debian member contacts an upstream developer, most of the time upstream developers are happy and proud to collaborate and help. The opposite is also true: upstream collaboration is highly valued in Debian, and is part of its ethical principles of building FOSS in a collaborative way, as stated in the Debian Social Contract.

Inner social interactions in Debian also have an important role: discussing online and in person during gatherings, not only technical issues but also about governance and voting issues, discussing the Debian Social Contract and all the tools that help and make possible these interactions between an internationally highly scattered community, are all seen as essential to Debian community existence and maintenance.

Tensions exist, and are visible during meetings or online discussions. Tools like codes of conducts, mediation, discrimination or violence prevention roles are developed and discussed. Authors Coleman and Hill described in their work on Debian [149] how the social production of ethics takes place, how it is reinforced through the sustained collaborative development of code, but also discussions and decisions around licenses and policies, and how in Debian in particular, this is structured around ethical volunteerism, institutional independence, and networks of trust between community members. Moreover, Coleman, by analysing three important ethical moments in Debian [148], shows how the ongoing and sometimes conflicting social re-evaluation of values in the community takes place, and how values related to accountability, freedom, transparency, openness, and mutual aid are moments of transformation and re-interpretation that help the community to continue functioning and working together.

Financial boosts in Debian Maintenance

The Long Term Support (LTS) consists of a 5 year long guaranteed support period of the current stable Debian releases for all architectures. This work is undertaken by the LTS team, but is financially supported by Freexian, an enterprise founded by a senior Debian community member and LTS team member, Raphaël Herzog, participant 13, that explained in our interview:

> "*Each Debian release is supported for five years: the Security team handles support for the first three years, and the LTS team for the remaining two, thus we have 5 years of guaranteed support on each Debian release. There is a lot of collaboration between the Security team and the LTS one: even though one team is primarily responsible for the first three years, the LTS team still helps out during those, and vice versa, the Security team gives us advice and provides feedback on the work related to LTS.*"

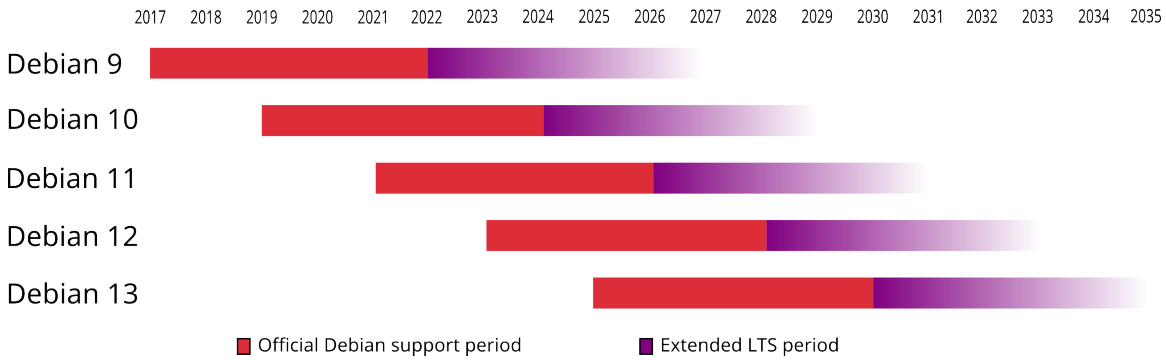


Figure 3.10: Debian LTS and eLTS periods for Debian stable releases 9 to 13 explained on the Freexian website.

Regarding the role that Freexian plays in the LTS work he explains:

> "*As you know a Debian release comes out regularly, without an established schedule, it goes out when ready, but in practice we have a new Debian release every two years. Historically we used to maintain these releases for only three years, by the Security team. Then Ubuntu came, a distribution derived from Debian, but offering 5 years of support. This put us on a positive concurrent position, and we tried to see if we could offer 5 years in Debian. The idea was to make enterprises pay for further maintenance work: if they used Debian and needed stable releases to be maintained longer, they could pay for it, and we could ask for their financial support. Freexian began to finance and manage the LTS work among Debian developers around 2014. Under the LTS model, Freexian collects a monthly amount from our enterprise sponsors, divides it into hours, and then allocates those hours to LTS contributors, who can specify the maximum number of hours they want to work. It's up to them to monitor the list of updates that need to be done and assign themselves specific tasks. Enterprises do not have their say in the work dispatch and management. This is important because, as Debian community and LTS team members, we know better what needs maintenance, what depends on what, while enterprises could focus only on visible packages, or what they directly need, and leave out important invisible ones.*"

Another maintenance system, called the extensive Long Term Support, eLTS, managed and financed by Freexian, aims at pushing the stable release maintenance up to 10 years, if financing allows to, as Figure 3.10, taken from the Freexian website shows.

Herzog explained:

> "*Unlike LTS which is an official Debian project hosted on the Debian servers and infrastructure, eLTS is not an official Debian project and is hosted on Freexian servers. Apart from the infrastructure, nothing really changes, it is exactly the same people involved. For the LTS we support all packages, the enterprise client does not choose. In the eLTS tier, it's the opposite: Freexian sets the price, and the customer defines the list of packages to be maintained. For example they can say I need security support for these packages that I have on my servers, and we tell them it will cost this much. The price we fix will change over time : the second year costs more than the first one and so on, because even*

> *if the support work is the same, the more the years pass the less demands we will have, so we ask more in order to compensate. From a financial standpoint, the eLTS tier is far more lucrative than LTS these days, we charge much more for that support: with LTS, enterprises can choose the amount they want to sponsor, so they don't pay much. With eLTS they have no choice: if they want support, they have to pay what Freexian asks for, and so in terms of revenue, it's much higher on eLTS than on LTS. Part of that money is reinjected back to LTS then, which receives much less financial support. Part of that money we also use for financing long term projects in Debian, invisible but important work that risks not being financed or undertaken otherwise.*"

To illustrate Herzog's final sentences, an important structural work has been undertaken in Debian—and in all Linux derived distributions these recent years—the usr merging work. It consists in merging two structurally and technically important system directories: the */bin* directory into the */usr/bin* one, the two having been separated for historical reasons not relevant anymore. Because this merge aims at being retro-compatible, i.e. done without breaking the system tools, but also all external tools that made the distinction between the directories, this is a huge work that needs to be assessed and discussed collectively. We discussed this work and the way it was undertaken with Helmut Grohne, participant 17, that was being financed partly by Freexian in advancing the matter. We also witnessed several community discussions on it.

The financing model of the LTS and eLTS programs, allows Debian to stay a volunteer-based community, keeping enterprise direct influence outside of it, while making them pay a part of the maintenance of a system that they use and need. This financing model was not implemented, as participants told us, without frictions and debates in Debian regarding the ethical values in the community. This solution was adopted, confirming the findings from Coleman [148], as a social re-evaluation of values in the Debian community as it re-structured itself in a self-preserving way.

3.4.3 Maintenance barriers in Debian

Maintenance obstacles in Debian are mostly technical and infrastructural. But the most feared ones are human and social factors regarding the community and its members.

An aging infrastructure and documentation

Carles Pina i Estany described how the aging documentation is sometimes a problem to maintenance work:

> "*To me one big problem of Debian is the documentation. Debian has evolved the last twenty-thirty years, documentation exists in many places, many ways and many variations that contradict each other. Different teams suggest different approaches so it's very hard to sit down, create and do, or even just know if this is the new way of doing packaging and not the other one. Having a sponsor who checks and tells you what to follow is very important in any case, but especially because of this.*"

Regarding the aging tools, he gave an example:

> "*I had some unit tests running in non standard ways in Debian, unit test from the upstream. In Debian we have a set of tools that help with this, but if upstream chose*

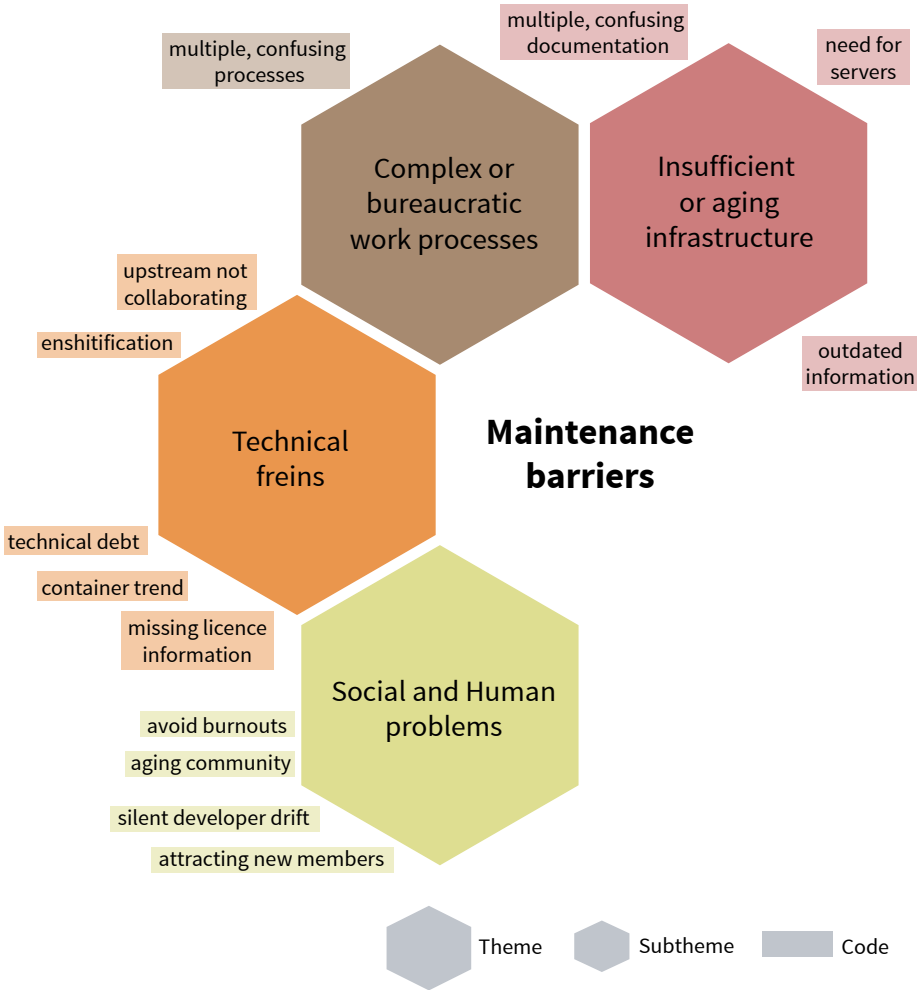


Figure 3.11: Maintenance barriers in Debian, as identified through thematic analysis.

another set of tools it is not always straight away compatible. So you need to add up certain things so these type of tests work. The Debian tool for this, called the debian-helper (DH) has many layers, it's built layer after layer after layer. If all works you are happy, but if it is not the case that is when you need to start scratching the layers to see what happens and it is not clear how to do it. Again, it is a 30 years old tool and it is not the most easy to use straight away."

When reflecting on solutions to the aging problems, Estany added:

"*Some of the problems we are dealing with is not packaging but because it is some 30 years old scripts and infrastructure, written in multiple scripting language, python, perl, and in multiple versions of them. To change these tools for more modern ones, it needs lots of backward compatibility because, we have maybe 30 000 of packages in Debian now that need to keep working and developers that do need to be able to know how to use new tools. It is old knowledge. But it is one thousand of developers doing it in their free time, so you cannot go and say well now we're changing all the way of packaging.*"

This last reflection from Estany on how to deal with aging issues, how to maintain, how to not

force upgrades but also take into account the human factor, the work processes and backward compatibility with other tools, appeared to us as something essential to the Debian project and values. Maintaining in Debian is not about forcing changes, even if this change is towards a better technical solution. Maintaining software is not just updating and upgrading. The updating - upgrading processes affect other software, they can break retro-compatibility, break development working processes, affect the work of the developers, and also of users. Maintaining in Debian is about analysing all the impacts of upgrading and updating, discussing the changes within the community and with the people affected by them in and outside the community, and trying to collectively find solutions, and taking the time needed for analysing the impacts, discussing them, and planning for change and adaptation.

This finding is, in my opinion, an important one of this study. In *The Labor of Maintaining and Scaling Free and Open-Source Software Projects*, Geiger et al. discuss the issue of developer burnouts in Debian as the package or tool they are developing scales up, and the need for maintenance increases, resulting in overwhelming number of messages, issues, bugs and notifications to deal with [82]. The authors describe the emotion toll of developers while engaging in technical work, but also in community or user support work. They analyse that maintaining is a process that has influence on people's labor and their reactions to them, affecting their long-term well-being in their communities. We will further discuss this in the next section (3.5).

Enterprise dark-pattern practices affecting maintenance in Debian

Ceratto (participant 8) described illegal telemetry and spyware features barriers in some upstream packages:

> "*Sometimes there are software that do polling: they connect to external websites, online services, to send telemetry, statistics, for advertising for example. We pace them out. Because in Debian we have policies where software should not create privacy breaches, have spyware-like features or dark-patterns like these, unless the user clearly wants this. This all should be opt in as the GDPR states, but Debian was careful about all this before GDPR, many years before, and we do not want them. The Lintian tool will tell us, it checks all sort of things with packaging. When we do this the upstream might complain that we do not have the right to remove spyware, it happened sometimes, but then if the package upstream is not cooperating on these we may decide not to package.*"

He also pointed out the following problem:

> "*Do you know the slogan: 'The single vendor open-source is the new close-source'? This happens more and more. The code is open but it is entirely controlled by a company that might do things to make it difficult to maintain as a fork, package, and in the end this is not real open source.*"

Ceratto described several examples that he had encountered following this pattern. One of them was about a database service company—ClickHouse—that developes an open-source product that they maintain only for a limited time, after what they ask for a payment for the support service. When trying to package the database and working upstream with them, the company directly told Ceratto that this was not in their interest, that their interest was to release frequently and support only for one year, and not support anything else that could go beyond (in this case Debian). To

Ceratto, this message was perceived clearly as a need for their product to be obsolete, so that they could sell support on it.

Ceratto feared that this trend is being generalized:

> "*You get some security updates, and then you have to pay to get more, and the update code is not being released. Even open-source companies do it now: Suse, RedHat have done it: if you pay you get priority fixes. And some fiwes may enter into the open source versions, some may never go there, or some may enter but later in time. Here in Debian everybody will agree that this is not safe, this is not security. I know It is an expensive work sometimes. But generally speaking the same company that is very careful with its own products, would never use something like what they sell. In their external products they purposely insert obsolescence.*"

Ceratto and Ferraris (participant 12), both members of the Mobian project, explained the problem with industrial actors not maintaining their hardware code, and providing it in blob forms or patches that are not upstreamed, not mainlined and little to no maintained. They detailed how this makes maintenance work at Debian very complicated. Ceratto:

> "*Sometimes upstreams do not fully cooperate and reject a patch that we provide for their code, but they still keep the patch locally and be careful not to break it when they change their code. This is still somehow good. Sometimes it happens that they do not want to, and Debian has to rebase the patch again and again every time they change their code upstream. For simple applications this rebasing is manageable. But we have some companies that are well known for releasing their own patches, never porting them to mainline kernel, so those who want to use that hardware have to maintain 200 patches. Impossible. For example kernel patches that are being decayed in Mobian. It is hundred of patches, rebasing all of them requires tons of work, making Mobian releases very difficult, that is also why Mobian is not yet an official Debian project, maintenance is too complicated.*"

Ferraris, head of the Mobian team, further explained:

> "*For Mobian we need to provide packages that can't be accepted into Debian, because they're, let's say, hacks, not optimal, since there are better ways to do things but they take more time. We maintain patches for certain packages that already exist in Debian, and would not be compatible with running the software on a traditional computer rather than a mobile device, because these are pieces of code that are more device-specific and alter the behavior compared to computers, but they perform better on mobile. But then we would lose functionality, ease of use, and ultimately quality of life on a desktop or laptop if we degrade them in Debian. So for the moment we keep these Mobian packages in a separate repository. Mobian, ultimately, will be a Debian system, to which we add a few Mobian packages on top to ensure it works well on the few mobile devices where Mobian runs.*"

Meanwhile, Ferraris and his team continue to work on integrating these separate packages into Debian. They had already been working on it for several years at the time of our interview in November 2024. For Bookwarm, the Debian stable release at the time of the interview, they had reduced the number of patches left to be integrated from 80 to 40, and for the next stable release six

months after, Trixie, they were hoping to cut it in half again. But once again, this was for maintaining very few smartphone devices in Mobian, less than ten devices at the time of this interview.

To summarize, we have identified through these quotations the following development anti-patterns that make maintenance difficult: addition of telemetry or spyware features, the excessive control that a company has on software (even if it is open source), selling updates and support as proprietary features, proprietary blob code, industrial actors not maintaining code, code that is never upstreamed, not mainlining the Linux kernel, patches are not a long-term solution in code maintenance. Many of these anti-patterns had already been pointed out in our previous chapter on the Android ecosystem.

The container trend problem

At the technical level, Ceratto highlighted something important about modern container trend systems, that make maintenance work for Debian, as well as for other Linux system distributions, more difficult.

> "*Static linking occurs when you build a binary and you statically link code to the libraries you are using, so these libraries are shipped as part of your binary code. Many popular languages like Rust or Go do this by design. There is a parallel to be made here with modern container trends, like docker or flatpak, that give you a simplified way to run an application on your system. Both static linking as well as flatpak and docker proceed in the same way: they throw everything inside a black box, and if there is a vulnerability it is very difficult to patch it. When you use a docker image, if there is a vulnerability, or a security update, you are not going to get it automatically unless you put it inside manually in the image, which very few users do. There was a paper where the researchers investigated in the last maybe five years the security of the contents of the docker hub and they found that more than fifty percent of the containers there had a significant amount of vulnerabilities that were not addressed. Flatpak is the same, shipping a lot of stuff in one blob. The problem is that these tools are more and more encouraging people to deploy this way, because it is easy, it looks shiny and modern, you do not need to bother to make efforts to be maintained the Debian way, perceived as heavy or old. As a consequence, developers care less about distributions, but this comes with lack of maintenance and vulnerability issues that are not addressed or taught to young developers as important maintenance practices.*"

To our understanding, these trend of encapsulating that Ceratto explained, operate some kind of shift of maintenance responsibility, from software developers to users, thus creating what we can call a maintenance debt, as users will in practice not be able to maintain the container boxes that encapsulate the software they use. This is maybe also something that has to do with two different ways of considering software in a system: Debian's way of maintaining is to deal with maintaining packages, and then deal with interdependencies between them at the system level. Containers isolate themselves from the system, as the idea is to be able to run them independently on every system: they act as small independent software systems, which is also where their value lies. But then the maintaining of containers is to be seen as the maintaining of an independent system, and to be compared to the maintenance of Debian, of Android OSes etc, not to that of a package.

Social and human maintenance barriers

Last but not least, our interviews highlighted an important concern in the Debian community: the maintenance of the social and human factors in Debian. How to avoid burnouts, attract new members and lower the entry barriers in an aging community, but also how to make inner interactions in Debian smoother? For Espany this was a primary concern:

> "*In Debian we have some 30 years old tools that some people are very happy to use, but new people have trouble using. For example mailing lists is not a communication system that young people like. This is a problem, but how to deal with it? Old versus new tools, people with different mindsets. It is very comfortable to stay with what you know, but it is not comfortable not to have new community members. What worries me the most is not the sustainability of the technical aspects, but of the human ones. How to attract and retain new contributors in Debian? How to increase the diversity of the contribution pool? How to prevent or deal with burnout of teams and people doing volunteering work?*"

Espany concluded our interview by quoting what he believes—though he is not certain—to be a statement by Ian Murdock, founder of Debian in 1993: *"Maintaining of Debian is not about interaction of packages, but interaction of people*". I was not able to verify this statement, but I understood it to be a major concern for him, and consistent with our findings in this study.

3.5 Discussing the Debian maintenance

OS maintenance in Debian is about long-term hardware support

Debian's maintenance process has multiple external positive impacts on the maintenance of other open source software. The first that comes to mind is the help in maintenance it offers to upstream packages. Most of the informants we discussed with, told us about sending patches and bug fixes to upstream code as the most important part of maintaining a package.

A second positive impact is that of the range of hardware architectures being maintained and ported to Debian, and the effort that the community puts on doing so. To quote the Debian official statement mourning the loss of its founder Ian Murdock, in 2016, *"Debian would go on to become the world's Universal Operating System, found on everything from the smallest embedded devices to the largest cluster systems, to the Space Station because "of course it runs Debian" which has been ported across multiple architectures and types of hardware*". We witnesses the importance of porting Debian to new architectures for the community during our different gathering participation. The new Debian ports were received with enthusiasm, and the community's interest in the porting process was very evident.

One of our first interviews was with Emanuele Rocca, participant 1, who had recently been able to port Debian to a Lenovo Thinkpad X13, holding a Snapdragon 8cx SoC based on ARM architecture. This announcement and the talk that Rocca gave during miniDebConf Cambridge 2023 on the subject made the community very enthusiastic. The joy of being able to port Debian, while explaining the technical challenges, was clearly palpable during our interview with Rocca. Debian's ambition of being ported to all kind of hardware systems has an impact on their obsolescence: by porting Debian into new hardware systems, once the initial vendor support stops—which we know

from the study of Android, could happen quickly—relying on Debian releases prevents the software-induced end of life of these hardware systems. During my experience in install workshops—helping people revive their old computers in local Linux User Groups (also called Linux install parties— I and my workshop colleagues have no hesitation in testifying to this reality. In these install partys we often deal with old computers that are unable to run on their initial OS, Windows or sometimes MacOs, either because no update is available anymore, or because the updated OS requires too much resources that the old hardware is not able to provide. And very often, by installing a Debian or other Linux OSes on them when it is available, these devices become functional again, running smoothly and capable of handling modern applications, while being updated to the most recent releases of the Linux kernel and Debian OS.

It is not a coincidence if numerous of these install parties present themselves as obsolescence remediation spaces: this reflects a recurring pattern in their activity. In November 2025, with the end of support of Windows 10, causing the obsolescence of more than 400 millions of computers worldwide[20] these install user spaces launched different European initiatives[21], for workshops where they helped the people to avoid Windows-induced obsolescence by installing Linux OSes.

What we notice here is an important difference in approach between Debian and some other widely used proprietary operating systems, such as Windows. When Windows 11 is released, hardware is being discarded. The same goes for OSes from Apple or even Android, when a new version is released, older hardware are and given lower priority than the new ones and often discarded. In Debian the effort is quite the opposite: when a new OS is released, hardware is not discarded. While new ports of Debian to new devices—not necessarily new as in recent on the market—are welcomed, old to very old hardware can still run on the newest OS versions, as illustrated in figure 3.2 and in numerous other examples that we witnessed in the community and during Debian install workshops. OS maintenance in Debian is about long-term hardware support, not a way of discarding them.

There is a relation here with what the community calls the 'universality' of Debian. From our observations this universality of the Debian system seems to be experienced in many ways: sometimes directly addressing hardware and software obsolescence, in other cases covering different use needs. For professional users it is a mark of reliability and stability at no cost, as participants n° 6 and n° 9, working for companies using Debian describe, supporting our observations of numerous company workers observed during the gatherings. For users of old hardware it is a solution to extend the lifetime of their devices. For developers and FOSS communities Debians serves as a basis brick to build derived operating systems covering different specific needs: security tailored distributions like Tails or Kali Linux, educational focused ones like Primtux, accessibility focused ones like Linux Mint or Emmabuntu, hardware based ones like Raspbian for Raspberry Pis, server focused ones like Yunohost etc.

I discussed the 'universality of Debian' and how it is understood, with debacle, participant n° 6, a Debian community member, working for Ammonit—an enterprise using Debian in its embedded wind and solar measurement hardware:

> "*You asked me why we chose Debian at Ammonit. Debian has proven stable through time, since more than 30 years now, that is also why we want to keep working with it. It is maybe not the best choice for each of our three use cases that we have in Ammonit,*

[20] See PIRG's summary of the obsolescence problem regarding Microsoft's Windows 11 upgrade: Why the end of support for Windows 10 is uniquely troubling, Sept. 2025, last accessed in April 2026.

[21] See the End of 10 initiative, or the Adieu Windows one, last accessed in March 2026.

but it does the whole work very well. We have this slogan, Debian the universal system, this idea is true in the case of Ammonit. Having Debian in the three situations is the best choice for us: developers use the system that they develop for and embed in the measurement boxes, they also use it on the servers that manage the boxes, so they better know what problems there are, they can better develop the tools we need, and collaborate with the Debian community to solve problems and add new features with their help, and then update and maintain the whole within the Debian community, who does it so well. By using Debian in the three situations, we also learn more, that is the main benefit in my opinion. We could use Yocto [a Linux distribution dedicated to IoT] for the embedded systems, but we would loose knowledge, because we would not be able to master everything inside of our systems."

Debian maintenance is not about enforcing update-upgrades: it is a socio-technical process of assessing how to implement them without causing harm

One of Debian's release mantras is *"We release when it is ready*", also called *ReleaseWhenReady* [22], meaning releasing only when the software meets the criteria for stability and usability that the community agrees on. The release process and time management has evolved during Debian's history [23], ReleaseWhenReady being the current Release Policy, even if it is debated and contested. In practice, Debian new releases in recent years follow a regular rhythm of every two years, give or take a few months. This time-freedom that the community decided to take in the beginning of the 2000s, when it realized that rushing to meet arbitrary deadlines compromised the quality, is quite surprising as it goes against what we observe in IT enterprise management doctrines, where time constraints, 'deadlines' and time-based management systems act as professional standards and put quality or worker health issues aside [162, 163].

Our study results suggest that this mantra is also about maintenance, as a process whose goal is not to technically update and upgrade tools, but about getting ready on the social and technical aspects for these updates and upgrades to take place. As discussed previously in 3.4.3 updates and upgrades can cause system instability, break retro-compatibility, affect work processes of developers, affect the users, and in turn create new maintenance needs in other technical or social aspects. These can also cause a shift of maintenance responsibility to users or other communities as we have already seen in the Android case by enforcing brutal changes, as Google did for example by enforcing eBPF (discussed in 2.4.3). The process of maintaining in Debian is about assessing the impacts of the changes, in different aspects at on different communities, discussing collectively the better way to achieve them, and taking the time needed for it.

This is also how Debian deals with support in the eLTS program. As Herzog (participant 13) explained:

"*In fact, we have modeled customer churn somehow, because enterprises that pay for the eLTS do not need ten years of support, often they just want a year or two to prepare and complete their migrations. So at the start we have a lot of customers willing to pay, but by the end of the ten years support that number drops, even though the maintenance*

[22] ReleaseWhenReady according to the Debian Wiki, last accessed in April 2026.

[23] Evolution of the Debian Release Philosophy: Release When Ready, Rune Slettebakken, August 2024, last accessed in April 2026.

> *work for a package remains the same. So, we charge more and more for each year of maintenance, which also encourages people to migrate, which makes sense for enterprises, because if it is not expensive enough, it's ultimately too easy for them to stick with aging architectures. Whereas the complexity of the maintenance work for us in Debian keeps growing: backporting a patch to software from five years ago is not easy to begin with, but on software from ten years ago, it's even harder.*"

This also appeared clearly at a conference talk from Evolix[24], an enterprise sponsoring multiple Debian events and whose founders and workers are closely involved in Debian. Evolix offers hardware and software hosting and support based on Debian to a wide range of enterprise clients. As their talk highlights, maintaining Debian systems for their clients is often a question of carefully preparing the migrations, in order to avoid discontinuities or instabilities, making sure that updating and upgrading will be smooth and well prepared, taking the maintenance time and effort it needs to be ready (even if the actual upgrading process takes less than a hour).

Maintenance is about collaboration

The Debian Social Contract (DSC), a fundamental document in the Debian community, presenting the values and principles of Debian, is a contract that Debian members make with the largest free software community. As shown in Figure 3.12, the contract emphasizes, especially in its second principle, the importance of giving back, distributing Debian's work freely (following the Debian Free Software Guidelines), *"communicating bug fixes, improvements and user requests to the "upstream" authors of works included in our system*" and *"placing Debian as a part of a bigger community, and of a bigger world where Debian plays its role*".

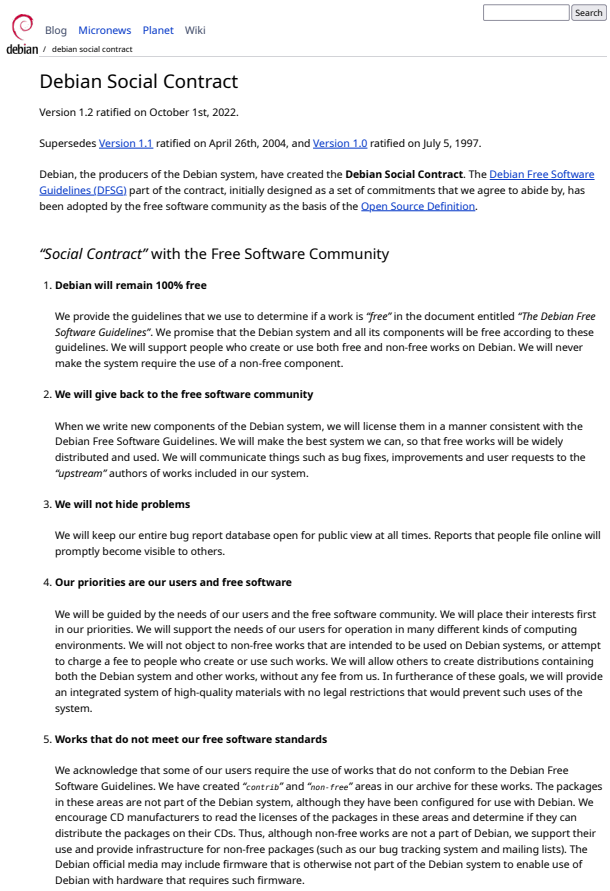
These community values are also reflected in the way maintenance is handled within Debian. For several years now, individual package maintenance in Debian has been pointed out as a weakness that could lead to discontinuity. Several community discussions have pointed out the resilience issues of individual maintenance, Debian past leaders have advocated to replace individual maintenance with collective one. As a result, package maintenance is increasingly being carried out by groups of people, usually within a team. This collective approach to maintenance tasks that Debian has implemented more strongly in recent years, fosters resilience: if one of the maintainers is unable to perform their duties, team members can step in to take over. This also involves mutual support, learning together, exchanging ideas and discussing maintenance issues, etc.

What this teaches us is that sustainability and maintenance in Debian is a collective process, within the inner community, but also into the wider community of FOSS, and a step further into the wider world of software and system maintenance.

Study limitations

The Debian gatherings that I attended all took place in Northern Europe, and thus the informants and participants in this study have not much diversity, and are most of the time Northern European. In Berlin I was happy to meet with a more diverse community of Debian members coming from Eastern European countries. The Western over-representation is a bias in the Debian community as well, not only in the gatherings that I attended. Overtaking this type of study in Debian communities

[24] Upgrading a thousand Debian hosts in less than an hour, Evolix talk in miniDebConf Toulouse, November 2024.



Blog Micronews Planet Wiki

Search

debian / debian social contract

Debian Social Contract

Version 1.2 ratified on October 1st, 2022.

Supersedes Version 1.1 ratified on April 26th, 2004, and Version 1.0 ratified on July 5, 1997.

Debian, the producers of the Debian system, have created the **Debian Social Contract**. The Debian Free Software Guidelines (DFSG) part of the contract, initially designed as a set of commitments that we agree to abide by, has been adopted by the free software community as the basis of the Open Source Definition.

"Social Contract" with the Free Software Community

1. **Debian will remain 100% free**

   We provide the guidelines that we use to determine if a work is *"free"* in the document entitled *"The Debian Free Software Guidelines"*. We promise that the Debian system and all its components will be free according to these guidelines. We will support people who create or use both free and non-free works on Debian. We will never make the system require the use of a non-free component.

2. **We will give back to the free software community**

   When we write new components of the Debian system, we will license them in a manner consistent with the Debian Free Software Guidelines. We will make the best system we can, so that free works will be widely distributed and used. We will communicate things such as bug fixes, improvements and user requests to the *"upstream"* authors of works included in our system.

3. **We will not hide problems**

   We will keep our entire bug report database open for public view at all times. Reports that people file online will promptly become visible to others.

4. **Our priorities are our users and free software**

   We will be guided by the needs of our users and the free software community. We will place their interests first in our priorities. We will support the needs of our users for operation in many different kinds of computing environments. We will not object to non-free works that are intended to be used on Debian systems, or attempt to charge a fee to people who create or use such works. We will allow others to create distributions containing both the Debian system and other works, without any fee from us. In furtherance of these goals, we will provide an integrated system of high-quality materials with no legal restrictions that would prevent such uses of the system.

5. **Works that do not meet our free software standards**

   We acknowledge that some of our users require the use of works that do not conform to the Debian Free Software Guidelines. We have created *"contrib"* and *"non-free"* areas in our archive for these works. The packages in these areas are not part of the Debian system, although they have been configured for use with Debian. We encourage CD manufacturers to read the licenses of the packages in these areas and determine if they can distribute the packages on their CDs. Thus, although non-free works are not a part of Debian, we support their use and provide infrastructure for non-free packages (such as our bug tracking system and mailing lists). The Debian official media may include firmware that is otherwise not part of the Debian system to enable use of Debian with hardware that requires such firmware.

Figure 3.12: The Debian Social Contract, as of April 2026.

in other continents, could have an impact on the results and findings, especially on maintenance barriers and at the social relations level.

I acknowledge that I am implicated in FOSS projects, as a volunteer, by advocating for them and contributing is socio-technical ways, without financial conflict of interest. This gives me a sensitivity to the subjects, a familiarity with the community members, but also maybe lack of distance. This has been something I have reflected upon, discussed with my thesis supervisor as well as researchers facing the same situation, and tried to consider when analysing the data.

3.6 Conclusion

In this chapter we studied software maintenance in Debian, a widely used operating system based on the Linux kernel, maintained by a non-profit volunteer community of more than a thousand of members worldwide, following the principles of collaborative free and open source development.

For two years, we attended four community gatherings, conducted formal and informal interviews with Debian maintainers, developers and community members, supplemented with participant observation, working sessions, analysis of technical literature and of online community spaces. We

analysed our data using thematic analysis.

Our findings show that maintenance work in Debian is located at different levels: at the technical level, at the organizational level, at the social level and at the infrastructure level. We show that Debian is structured around maintenance work, and that this maintenance work is a socio-technical process. Our study details the role of social maintenance in Debian: the social relationships between Debian members and external upstream developers play an essential role in the maintenance process, while the socio-technical relations inside the community, sustained by a large infrastructure of tools, documentation and software, help maintain the community. Maintenance in Debian is not just technical, it goes through effort in maintaining social relations, community infrastructure and continuous collaboration among actors inside and outside of the system. Sustainability within the community: resolving conflicts, avoiding burnouts, attracting new members, creating an inclusive and diverse techno-social environment, retaining members, seem to be important concerns for maintaining the social foundations.

Our findings underscore the importance of the collective process of maintaining. Maintenance in Debian is not about performing technical updates and upgrades, but about taking the necessary time and effort into preparing systems to get ready for them, by avoiding ruptures and thus obsolescence. It is a collaborative socio-technical process of assessing, discussing and preparing the best approach to perform the update-upgrades.

Our study highlights the impact that maintenance in Debian has on the maintenance of software and hardware broadly speaking. At the hardware level: OS maintenance in Debian is about supporting old hardware, not discarding them. It is also about indiscriminately supporting new architectures, not only the most sold ones. At the software level: maintenance in Debian positively affects the maintenance of other free and open source software or OSes, but also more broadly, of all hardware-ecosystem that use Debian.

These findings highlight the importance of exploring the concept of maintenance in the light of Debian practices, offering an alternative narrative on software maintenance and on obsolescence management.

Chapter 4

Obsolescence: a business model of digital capitalism

Contents

It is common to attribute cases of obsolescence to computer-related causes: a chip that blocks a printer, an operating system that reduces battery performance, software that prevents repairs. In French, the term *programmed obsolescence*, a mistranslation of planned obsolescence from english, suggests that the model of obsolescence is a computer malfunction, pre-programmed by an internal mechanism. In France, the 2015 law that made it a criminal offense was amended in 2021 to simplify it, but also to emphasize this aspect: Law No. 2021-1485 defines planned obsolescence as *"the use of techniques, including software, by which the party responsible for placing a product on the market aims to deliberately reduce its lifespan"*. Complaints filed in France have targeted smartphones, video games or printers[1].

Digital technology would thus be a field particularly prone to obsolescence, and a means of causing it. This observation, often made with a critical intent, fosters a fatalistic stance: if obsolescence arises within and through digital technology, isn't it inevitable? Critics and defenders of obsolescence sometimes agree on a single idea, one that has become commonplace in the media and government

[1] the iPhone obsolescence case: hop.org, Feb. 2020; the Ubisoft video-game obsolescence case: quechoisir.org, Sept. 2020; the HP printer obsolescence case: halteobsolescence.org, Nov. 2024.

communications: the digital world changes quickly, *"everyone knows that"*[2]. We are experiencing a *"digital revolution"* an *"inevitable disruption"* that requires us to *"reinvent ourselves"*[3]. In this scenario, obsolescence is the collateral damage of progress that cannot be stopped, whether we lament it or not, as in the *Creative-Destructive* innovation ideal that economist Schumpeter coins—the idea that new methods of production survive by eliminating existing ones, creating the conditions to growth in capitalism and the free-market(better products, greater profits), but rely on destruction and obsolescence to achieve it [68]. Historians, and some economists like Schumpeter himself, have questioned this approach, criticizing the notion of *transition*, pointing out that innovation introduces friction within a system, can cause loss of quality or functionality [66, chapter 7], and even delays in the maintenance and repair of technical infrastructure [164].

In this chapter, we will examine this view of the history of digital technology, in which the accelerated replacement of each model by a new one is seen as progress, and the definitions of the obsolescence that underlies it. Far from being a technical phenomenon or a natural *law* governing this sector, obsolescence is a matter of planning—as the English expression *planned obsolescence* acknowledges. To understand how obsolescence has established itself as an economic model and a strategy for organizing the IT (Information Technology) industry, we will revisit the process of hardware miniaturization, and the dynamics of privatization of software.

4.1 First the hardware: from miniaturization to cloud computing

4.1.1 Miniaturization and planned obsolescence

The history of computer hardware production, from the first scientific calculators to personal computers, has been marked by miniaturization. The manufacture of integrated circuits—or chips—along with the new possibilities of mass production of these chips during the 60s, made it possible to reduce the size and manufacturing costs of computers. In 1971, Intel assembled all the electronic components needed for a processor onto a single silicon wafer, microprocessors, which themselves would enable the production of microcomputers [165]. But these products remained expensive, and demand was limited. Intel's subsequent success is due to a production plan, later dubbed '*Moore's Law*' after Gordon E. Moore, Intel's co-founder. This formula is meant to capture the pace of innovation in the field of microprocessors and printed circuits: the power and integration of microprocessors increase at a steady rate, while costs decrease. However, as Sacha Loeve writes, a reading of Moore's articles and the history of their dissemination show that the 'law' is in reality the result of deliberate planning for the obsolescence of his products [166, 167].

In 1965, while serving as director of R&D at Fairchild Semiconductor, Moore published an article [168] in which he argued that it would be profitable to mass-produce integrated circuits for the civilian market. At the time, cathode-ray tubes dominated the industry, and his company worked primarily for the Unites States National Aeronautics and Space Administration (NASA). According to Moore, a margin over production costs could be maintained by gradually increasing the complexity of integrated circuits in a mass-production scenario for the civilian market. The question remained: how fast should this increase occur? At what point should an innovation be

[2] *2045: the year man becomes immortal*, Lev Grossman, Time.com, 10 Feb. 2011.
[3] The white paper on digital revolution, Syntec Numérique, last accessed in April 2026.

introduced to renew the economies of scale enabled by mass production? Moore proposed doubling the complexity of integrated circuits every year.

At the time, this proposal went unheeded. The first to apply it was Moore himself, in 1971, shortly after he co-founded Intel and launched the first microprocessor, the Intel 4004, which would enable Moore's plan to become profitable. New models would be regularly released, incorporating more and more transistors, though at a slower pace than what was proposed in 1965.

In Intel and Moore's announcement, this plan—which was later (and imperfectly) implemented—was presented as a prediction that was retrospectively (and perfectly) verified, that they called a 'law'. In a 1975 article, Moore describes the history of integrated circuits as one of *"exponential growth"* and *"progress"* in *"new technologies"* [169]. He announced that the rate of circuit integration would henceforth double every two years, a *"slowed pace"* that nevertheless attested to a *"trend"* or *"pace of progress in electronics."* The same ambiguity between predicting and provoking appears in an article in 1997, where he argues that the rate of integration has doubled *"every eighteen months or so"*, constituting an *"exponential growth"* characteristic of *"new technologies"* [170]. Moore thus presents the result of his own choices as facts without an author, his propositions as predictions, even as he constantly modifies the pace (one year, two years, eighteen months) and the units of measurement (cost-per-integration-density ratio, MIPS). As the head of the company dominating the sector, he had the means to de facto impose production standards and the language used to describe them. Moore's articles, originally management ones, became communication tools, adopted by various players in the digital sector.

Indeed, as Loeve analyzes, the term *Moore's Law* is often used to promise the multiplication of something by a constant factor on a periodic basis. Approximately every two years since the 1970s, microprocessor manufacturers have promised smaller chips. Starting in the 2000s, the nanotechnology sector has used this rhetoric to secure public and private funding, promising quantum computers for everyone, which 25 years later are still not here [171]. As Loeve points out [166, p. 100-102], the International Technology Roadmap for Semiconductors (ITRS), renamed as the International Roadmap for Devices and Systems (IRDS)—a consortium of semiconductor manufacturers—has produced an annual *"roadmap"* since 1968. One can read on the official website that the roadmap is intended to serve as *"the primary reference for the future of university researchers, consortia, and industrial researchers to stimulate technological innovation"* [4]. In 2005, it announced that *Moore's Law* would be surpassed thanks to two new roadmaps: *More Moore* for component miniaturization, and *More than Moore* for the diversification of applications (Internet of Things, biotechnology, etc.). These two aspects converge toward System on Chips (SoC), as illustrated in figure 4.1 taken from the official roadmap document.

The *More Moore* roadmap is a continuation of the miniaturization: in 2020, three major manufacturers announced they were producing *"5-nanometer (nm)"* chips, a new target following the *"7 nm"* announced in 2018, preceded by *"10 nm"* in 2016, *"14 nm"* in 2014, and so on since 1961[5]. Every two years, these ITRS roadmaps schedule the release of new semiconductors that are more miniaturized than the previous ones, and manufacturers follow. Thus, in 2022, TSMC (Taiwan Semiconductor Manufacturing Company), the industry leader, announced it had achieved *"3 nm"*, then in 2025, *"2 nm"*, and the IRDS projected the *"1 nm"* for 2027.

It is no surprise that this planning is unfolding so smoothly, given that behind these nanometers,

[4] From Moore's Law to NTRS to ITRS to IRDS, last accessed in May 2025.

[5] See the MOSFET scaling in the Semiconductor device fabrication diagram, Wikipedia, last accessed in April 2026.

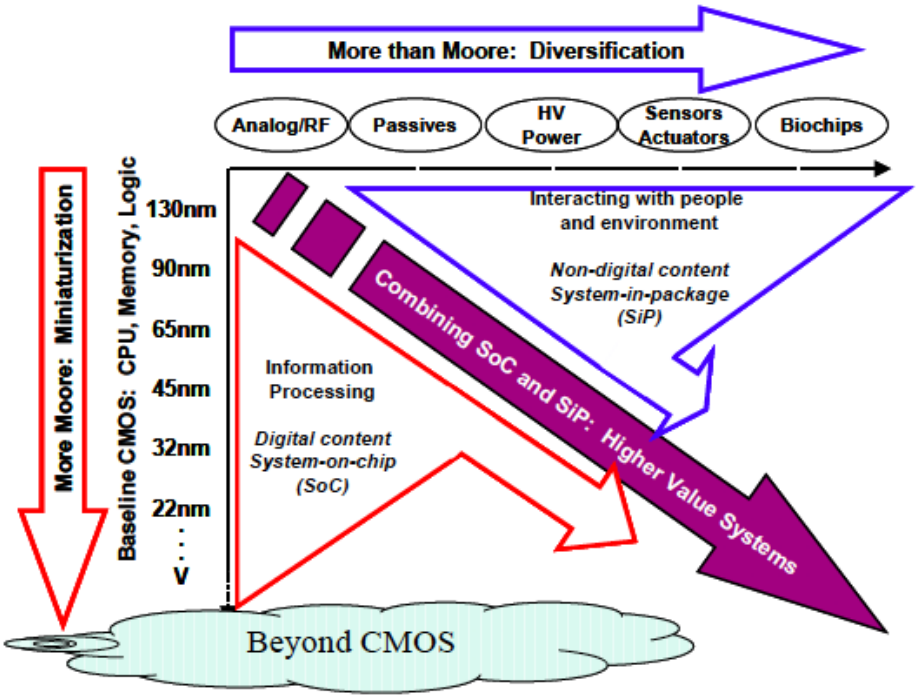


Figure 5 Moore's Law and More

Figure 4.1: Figure taken from the IRDS roadmap on *"Moore's law and more"*. Source: irds.ieee.org, last accessed in April 2026.

we no longer find innovations in miniaturization but a marketing label. Indeed, as early as 2009, 'X nanometers' no longer referred to the size of chips or a technical process, but a product marketing model. TSMC's vice president acknowledged this himself in 2019 [6]:

> "*Today, these numbers are just numbers. They're like models in a car— it's like BMW 5-series or Mazda 6. It doesn't matter what the number is, it's just a destination of the next technology, the name for it. So, let's not confuse ourselves with the name of the node with what the technology actually offers.*".

These entrepreneurs are in fact reviving a model introduced by General Motors (GM) in the 1920s, which involved numbering products to give the impression of constant performance improvements [59]. They are thus very much the heirs of Moore and his ambivalence: they plan their production as a series of periodic updates, while claiming that this plan is the result of a 'law'. Others have attempted to model obsolescence, such as scholar Peter Sandborn (as we previously discussed in the introduction 1.3.1), who has also worked with industries and companies to help them predict and avoid parts and software obsolescence in their internal production processes. The French Institute of Obsolescence (IFO), a branch of the International Institute of Obsolescence Management (IIOM), is dedicated to discussing the best methods for *"managing obsolescence"* for industrial production. During their *Obso-Days* annual conference, so-called *"obsolescence experts"* present it as a *"vector of agility"* and a *"lever of sustainability"* [7].

[6] Philipp Wong, in , 10 Sept. 2019, last accessed in April 2017.
[7] See IFO's Obso Days 2025, Institut-obsolescence.info, last accessed in Dec. 2025.

Meanwhile, big tech executives put obsolescence in practice for consumer goods they companies sell. Bill Gates, founder of Microsoft, declared in 1999: *"In three years, all the products my company makes will be obsolete. What matters is whether we make them obsolete, or whether someone else will do it for us*" [55, p. 152]. Obsolescence is not only admitted, but presented as inevitable. Similarly, Raymond Kurzweil, a futurist who became Google's chief engineer in 2012, wrote in 2005 that 'Moore's Law' is a law of evolution governing biological, technical, and cosmic development [172]. Obsolescence is presented as an independent phenomenon from which one can profit provided one assumes it and anticipates *"technological trends*" [172, p. 3].

Supporters of transhumanism, *effective accelerationism*, big tech executives and the mainstream media have taken up these ideas [173] [8]. They have become commonplace in corporate communications: in 2024, Jensen Huang, CEO of Nvidia, gave a speech at the *COMPUTEX conference* in Taipei. In a room filled with 6 500 IT professionals, he explained that *"the future of computing is accelerating*" and that *"With our innovations in AI and accelerated computing, we're pushing the boundaries of what's possible and driving the next wave of technological advancement*" [9]. In reality, the event was intended to unveil a new Nvidia *"semiconductor roadmap*": *"Our company has a one-year rhythm. Our basic philosophy is very simple: build the entire data center scale, disaggregate and sell to you parts on a one-year rhythm, and push everything to technology limits*". Every year, a 'new generation' of data center graphical chips purported to be more powerful, cheaper, and more energy-efficient. Huang's conclusion: *"the more you buy, the more you save*".

This business model for organizing the ongoing repurchase was the same as that of Moore for Intel or of General Motors: a roadmap for new models on a *"one-year rhythm*" announced at a high-profile media event, during which the models of the year are unveiled to an audience already convinced they are witnessing the privileged unveiling of a historic event. Nothing new, then, in this event-driven marketing, whose role is to sustain a performative discourse over time: announcing the imminence of a new era within a ritualized context, produces a vision of history which, because it is disseminated and shared, imposes itself as the very form of social time. In France, despite Moore's approximations and rewritings, his 'law' is still taught to computer science students. I recall precisely how teachers presented Moore's law to me in my first university years, as a computational property of electronic materials. In a specialized master's program offered by the Paris Institute of Mechanical Engineering and title *Sustainability-Obsolescence-Scarcity* obsolescence is a variable to be controlled, slowed down, or accelerated depending on opportunities.

Thus, obsolescence in the digital realm guides production in a manner that is both deliberate—it is planned and these plans are published—and disguised—it is presented as a law of nature. Strategic obsolescence is obscured by narrated obsolescence. This naturalization of industrial choices is a constant in the history of obsolescence and can also be found in the history of the textile industry, architecture, or machine tools [72].

4.1.2 Embedded systems: new vulnerabilities

The history of miniaturization also includes the widespread adoption, beginning in the 2000s, of embedded systems connected to the Internet via cloud computing. As the fundamental physical

[8] The rise of end times fascism, N. Klein & A. Taylor, the Guardian, April 2025, last accessed in April 2026.

[9] 'Accelerate everything' NVIDIA CEO Says Ahead of COMPUTEX, Nvidia blog, June 2024, last accessed in April 2026.

building blocks of smartphones and the Internet of Things (IoT), these systems present new risks of obsolescence and digital dependency.

The main component of embedded systems is the SoC, a single chip containing all key components integrated side by side: computing or graphics processors, memory, and sensors. SoCs also integrate embedded software or firmware, making them autonomous systems ready to be embedded in any object, thereby digitizing it: cars, refrigerators, washing machines, light bulbs, and even connected water bottles [1] (see also illustration in figure 4.2). The ultra-fine soldering of components saves space, reduces energy consumption, and enables the creation of battery-powered portable devices. In a smartphone, the SoC is a key component, comprising one or more microprocessors, memory, one or more graphics processors to handle the display or AI computations, modems (cellular network, Wi-Fi), and sensors (Bluetooth, infrared, biometrics, etc.). On top of this are the peripheral components (touchscreen, cameras, speaker, microphone), all of which depend on the SoC, which manages them and handles interactions with the user. This lack of modularity creates new hardware vulnerabilities for devices with SoCs: replacing a faulty component of the SoC is physically almost impossible, even for professional repair technicians. The embedded software in the SoC, provided by the manufacturer, adds a layer of dependency and lack of interoperability, creating a strong dependence of users on manufacturers, as code and documents from these software are almost never revealed by the manufacturer. Thus, the software layer embedded in the SoCs creates a new set of device vulnerabilities, which makes repair or do-it-yourself (DIY) practices even more difficult. Yet these are important strategies for addressing obsolescence, extending the lifespan of devices, maintaining them, caring for them, or even expanding and repurposing their use [106, 174].

SoCs are not the only reason for the difficulties in repairing the devices that contain them, such as smartphones, tablets, connected watches, etc. Repairing smartphones and other connected devices is all the more difficult and expensive because their design makes disassembly complex and risky. The Right to Repair Coalition has previously criticized Apple and Samsung for designing non-removable battery systems: in 2007, the first iPhone was sold with a soldered-in battery that only Apple service centers could replace [10]. According to the company and enthusiastic media outlets, the non-removable battery and the requirement to have repairs done at Apple are the conditions necessary for a slim phone protected from water ingress. Other manufacturers have followed this model, which has become the de facto standard. But 'de facto' does not mean 'by itself', without persistent intervention by interested parties. This becomes evident when civil society opposes them: we then see lobbies defending the model established by the company. The European regulation on batteries and battery waste, adopted in July 2023 [11], came too late and has not yet remedied this situation, according to the Right To Repair EU coalition [12].

Another example is that of the universal charging port USB-C. It wasn't until 2024 that EU regulations requiring it for smartphones, tablets, and other portable devices came into effect—a topic that had been discussed since 2009 but was subject to significant lobbying. Similarly, the slim design of devices, supposedly demanded by the public, makes the hardware more fragile. In 2014, Apple was embroiled in the *bendgate* scandal: the frames of the iPhone 6 and 7, too thin, would bend

[10] Has the Age of Self-repair arrived? Access to smartphone parts may be improving, but smartphone design is not, April 2023, repair.eu, last accessed in April 2026.

[11] Regulation (EU) 2023/1542 of the European Parliament and of the Council of 12 July 2023 on batteries and battery waste, eur-lex.europa.eu, permanent link.

[12] Making Batteries Removable and Replaceable: a closer look at the new EU Guidelines, bad news for users, Repair.eu, February 12 2025, last accessed in April 2026.

Figure 4.2: Picture of an illustration of existing connected devices using STMicroelectronics chips, from a personal copy of the book *"Toujours puce"*, from Lecarpentier Elsa & Maud [1].

in users' pockets. Despite Apple's denials, internal documents show that the company was aware of the issue [13]. Yet it continued down this path: in 2025, Apple launched the ultra-thin iPhone Air, while the advertisement for the Samsung S25 in the same year declared it *"beyond slim"* (*"the thinnest of the thin"* in the French version).

According to Cory Doctorow, Apple *"treats drops, slips, cracks, or battery aging as features, not defects"* [14]. The expression *"it's not a bug, it's a feature"*, a quip originating in the world of software development, has become a meme mocking these so-called innovative defective products [15]. Doctorow details other strategies that Apple have used and continues to use in order to control the market and hinder repairs: the use of patents or trademark law; weakening repair laws; contractual agreements forcing shredding instead of repair [16] to name just a few.

[13] Apple Confirms It Knew About iPhone 'Bendgate', 24 May 2018, last accessed in July 2025.
[14] Apple's Cement Overshoes, Cory Doctorow, pluralistic.net, May 22 2022, last accessed in April 2026.
[15] It's Not a Bug, It's a Feature, Nicolas Carr, August 19 2018, wired.com, last accessed in April 2026.
[16] See Apple Forces Recyclers to Shred All iPhones and MacBooks, Vice, April 29 2017 and Apple iphone recycling program has secrets, Bloomberg, 18 April 2024, last accessed in April 2026.

The increasing amount of these devices in recent years, thus creates new e-waste. As an example of the innovation and its *Creative-Destructive* ideal, the launch of Apple's AirPods in 2016 was accompanied by the disappearance of the 3.5mm audio jack on smartphones, rendering wired headphones incompatible. A forced transition to wireless earbuds has taken place over the past few years from every phone company (including Fairphone), thus shifting from a simple standard to an embedded system containing a dedicated chip and a non-removable lithium-ion battery, at a much higher environmental cost. During our visit at Loxy, the WEEE treatment center in the Île-de-France region[17] reported to us that they have noticed an increase in waste in recent years of embedded disposable devices such as e-cigarettes, Bluetooth earbuds, or connected surveillance cameras, each containing a lithium battery and one or more electronic chips. In figures 4.3a and 4.3b we show some pictures of the recycles bins at Loxy filled with them.

(a) Disposable e-cigarette waste bin. (b) Connected surveillance cameras waste bin.

Figure 4.3: Waste bins filled with IoT devices at Loxy WEEE treatment center.

4.2 The great privatization: obsolescence through software

The hardware-related obsolescence strategies are not the only obstacles to extending the lifespan of digital devices. Obsolescence opportunities can also be software-related.

4.2.1 Software enclosure

Until 1969, computers were sold with an operating system included in the purchase price at no extra cost. In 1969, IBM separated hardware and software for the first time, offering the software as a separate component in its sales offering for the System/360 (S/360), a series of interoperable and

[17] Newsletter n°6 of the Limites Numériques team: End of life & digital waste, March 14, 2023, last accessed in April, 2026.

modular computers. Thanks to its modularity and hardware and software interoperability, the S/360 series became an industry standard within a few years, making IBM the market leader in professional computers [165]. Following a lawsuit for abuse of a dominant position, IBM's competitors won the right to sell or lease S/360-compatible computers. IBM then decided to unbundle its software offering from computer sales and accompany it with a copyright-licensing system ensuring its full and exclusive ownership of the software [175]. This maneuver allowed IBM to maintain its dominant position in the computer market. Watts Humphrey, director of systems and applications at IBM, was a member of the working group responsible for this unbundling, composed of lawyers and technicians. According to his own words [176]:

> "*Our charge was to recommend how to unbundle software, not whether it should be unbundled. We debated a family of asset protection alternatives, including patenting, trade secrets, and copyright. [...] It was not clear that software could be patented, and the volume of software products and versions would quickly overwhelm the patent process. [...] While we viewed copyright as a weak form of protection, it was all that we had. To improve the level of protection, we coupled the copyright with a license and counted on the license to provide the real protection.*"

The separation between hardware and software was not a technical consideration, it was a business decision, aiming at creating a new software market, according to Humphrey. Looking back, as a software and system developer, he considers it to be a bad decision: *"By unbundling systems programs, we have imposed an artificial barrier to the advancement of technology*". Once again, we reproduce here the 'Creative Destructive' aspects of innovation: innovation is not about improving, it is about replacing products by planning their obsolescence.

This economic-legal model of creating a new business market for software, using the law—patterns, copyright trade secrets in this case—is a process very similar to what legal scholar Katharina Pistor describes in her book *The code of Capital: How the Law Creates Wealth and Inequality* [177]: *"With the right legal coding, any object, claim, or idea can be turned into capital*". Pistor takes several examples to illustrate: the inclosure act enclosing common-land in 1600; indigenous land appropriation in Western colonialism; intellectual property rights enclosing art or ideas. Pointed out by scholars like Boyle [178] as the enclosure of *"the commons of the mind*", or Aigrin [179] and Maurel [180] that have expanded it to digital commons as the *"enclosure of knowledge commons*". We argue that the unbundling of software from hardware follows the same logic of creating capital and inequalities by enclosing software.

This law-coded software enclosure—unprecedented at the time—was adopted by a fledgling Microsoft in the 1970s. Microsoft designed all its operating systems, including Windows, as separated from the hardware, marketed under a single-use copyright license that granted no access to the source code, permitting neither modification, copying, nor reverse engineering. Intellectual property rights amplified the user's dependence on their own machine, first through the operating system, and later through proprietary programs, such as the Microsoft Word office suite, which could only be installed on Windows, the data it produced were a closed format decided by Microsoft, and whose paid, single-use license expires frequently.

Microsoft also practiced software-hardware tying—under a commercial agreement between Microsoft and manufacturers, the sale of a computer is tied with a Windows license—forcing users to purchase both products together, and thereby establishing Windows as a de facto standard oper-

ating system. This practice, dubbed *racketware*, although it is repeatedly denounced by consumer advocacy groups, is now the de facto standard of selling electronic devices.

This software-induced market domination allows Microsoft to create a captive market and force massive equipment upgrades. Thus, Windows 11—released in October 2021 as the successor to Windows 10—is incompatible with all processors manufactured before 2018. This affects, according to estimates, one-quarter of the installed base, or more than 400 million PCs. By announcing the end of support for Windows 10 in October 2025, Microsoft forces obsolescence on user devices—a waste of resources that has been widely condemned in many countries.

These software-induced de facto standards do not result from 'trends' but from choices: the choice of consumer capture models, which are disseminated through competition and supported by technical, legal, or political forms of coercion.

4.2.2 Smartphones and connected devices: accelerated obsolescence

The software layer embedded by design in SoCs is subject to the same processes of business-oriented lock-in and control by companies dominating the sector. The discontinuation of updates and maintenance, aided by the shift to cloud computing, leads to abrupt disruptions in usage and a faster pace of device replacement. Android smartphones are a prime example of this dynamic as we already discussed in chapter 2.

These issues affect all devices with embedded systems. In 2023, Arlo announced that its video surveillance cameras would cease to receive updates four years after their release, due to *"a number of reasons, including changing market demands, technology innovation, development of alternate and more efficient software platforms, and/or improvements in product and cloud security*". And the company began offering discounts and purchase coupons to *"encourage*" consumers to buy a newer model [18].

More recently, Google announced the end of software support for its Nest connected Thermostats of 1st and 2nd generation in October 2025. The thermostats would no longer receive software updates *"which may lead to decreased performance with continued use*". It would not be able to connect to the Google servers anymore or to The Nest application from Google, but *"users will recive 50% discount to purchase the latest tado° thermostat*"[19]. In November 2025, hacker Cody Kociemba reverse-engineered the thermostats to restore their functionality—a practice prohibited by law—while hacker Team Dinosaur developed an open-source software to make the thermostats regain the main lost functionalities. Their work was financed by the *FULU Foundation*, a non-profit acting for Digital Ownership Rights [20].

Changes in features, access conditions, or abrupt service terminations by companies remotely controlling connected devices have become commonplace. The Public Interest Research Group—a US & Canada consumer rights non-profit—has compiled a non-exhaustive database of such cases. Called the *"Electronic Waste Graveyard of expired software and connected services*" it estimates that in 10 years since 2014 these practices have generated 45 million kilograms of electronic waste worldwide[21]. This was before the end of Windows 10 support. These are strategies of what Cory Doctorow

[18] Arlo will end support for these older cameras in April, zdnet, Jan. 2023, last accessed in April 2026.
[19] End of support for Nest Learning Thermostats, support.google.com, last accessed in April 2026.
[20] This Group Pays Bounties to Repair Broken Devices—Even If the Fix Breaks the Law, 12 Dec. 2025, Wired, last accessed in April 2026.
[21] Electronic Waste Graveyard, PIRG, last accessed in April 2026.

has called the *"enshittification"* of digital technology, be it devices, software or services [181]: a process in which connected devices, online products and services decline in quality over time. While these practices are denounced and subject to laws and lawsuits, and consumer protest movements are multiplying, they continue to be widely used by companies [182].

4.2.3 The open source hardware movement

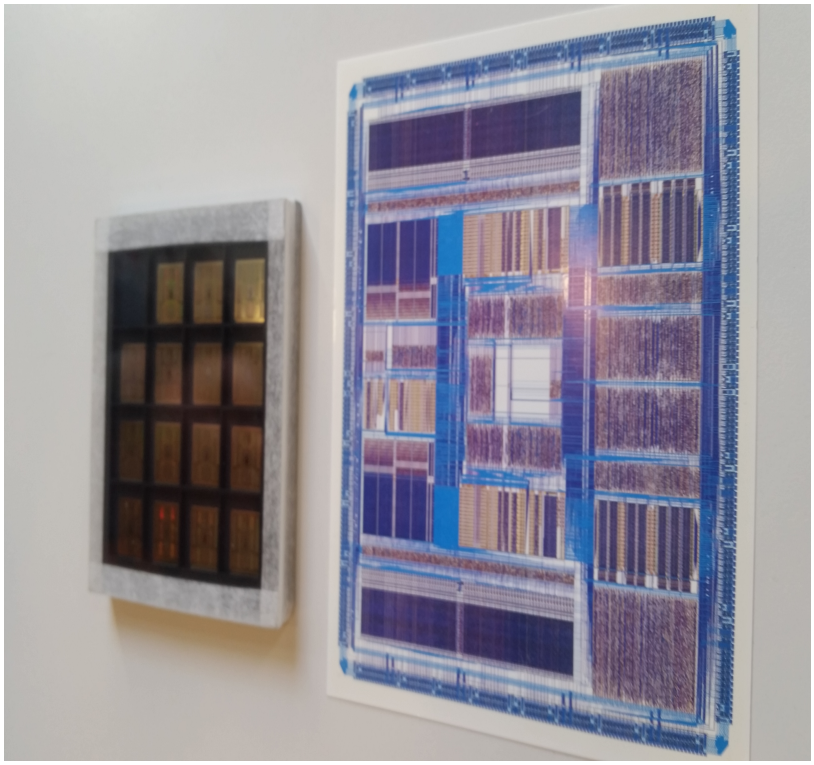


Figure 4.4: Picture of an open hardware SoC schematics and prototype from the ALSOC team at LiP6, provided by Marie-Minerve Louërat.

During this PhD, in several occasions, I met with researchers and FOSS community members working on the issues of open source hardware. The open source hardware movement—not very well known compared to its software counterpart—advocates for open standards, free and open source software and open documentation of hardware design and manufacturing.

At the research team ALSOC of the *Laboratoire d'Informatique de Paris 6* (LIP6), researchers develop open source software and design techniques for open SoCs. In figure 4.4 I show a picture of the design schematics and a prototype of one of their open hardware SoCs. In 2024 the team hosted the annual Free Silicon Conference (FSiC2024) with a special session on sustainability[22].

Another example of open source hardware is that of the MNT Reform computer that was described to me by Johannes Schauer Marin Rodrigues (participant n° 4 (table 3.1) at a Debian gathering in Berlin. The MNT Reform is an open hardware laptop, modular, repairable, functional[23].

[22] Free Silicon Conference Sustainability session FSiC2024, last accessed in April 2026.
[23] MNT reform open hardware laptop, official website, last accessed in April 2026.

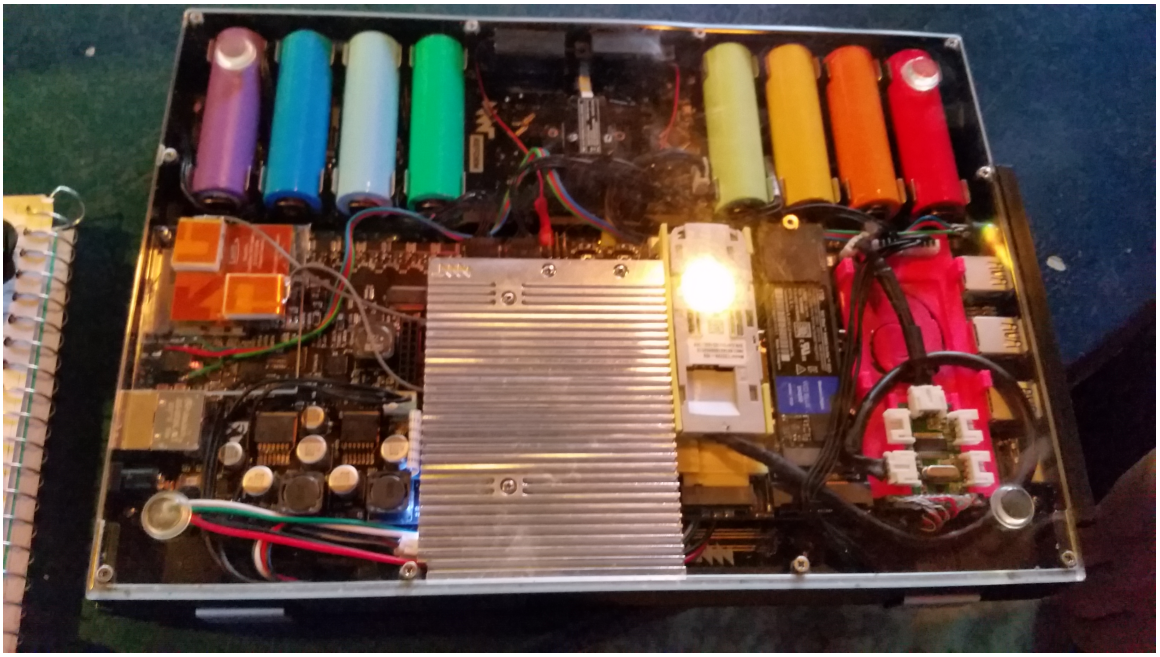


Figure 4.5: Picture of the MNT Reform open hardware laptop of Johannes Schauer Marin Rodrigues.

During my research on Debian, Johannes Schauer Marin Rodrigues explained his work on the Debian port of MNT Reform computer (4.5) and the importance of open hardware:

> "*It is very difficult today, very few projects have open source hardware. The MNT reform is as open and transparent as possible, not completely because it is not yet possible, but this is definitely one of the most open hardware that one can get from a laptop. Some of it's hardware and firmware, all software and documentation is open source: the firmware which drives the system management controller, the design files for the motherboard and system-on-module are open. It is modular, very easy to change and adapt the components, all is documented. I develop the Debian port, and it is today the main official OS on the laptop.*"

The interest of these open hardware lies in the highly repairable, hackable hardware but also open source software that facilitates maintenance.

Although it remains a niche area—few people are developing or buying such products, as the challenges involved in designing, let alone manufacturing, are enormous—it nevertheless demonstrates that open hardware is possible and is a solution to the problems of the lack of updates from closed firmware, drivers and frankenkernels.

4.3 Cloud computing: permanent connectivity, disposable infrastructure and disposable data

The widespread adoption of embedded systems is made possible by the *cloud*: software delivered as IT services, no longer running on local computers but on remote servers, located in data centers, connected to the Internet. The *cloud* has existed since the early days of the Internet, when websites

needed to be hosted on servers. However, the term gained widespread use as the use of remote services expanded to all sorts of previously disconnected applications: listening to music, watching movies, office automation, document and media storage, to name but a few. This includes connected devices, which rely on the cloud to function and become obsolete as soon as the corresponding cloud service is turned off.

This process, which we propose to call *cloudification* is accompanied by constant connectivity and a new form of *data disposability*. Before *cloudification*, an Internet connection was a temporary, provisional state—requiring active intervention by the user to establish and terminate it—and was perceptible—accompanied, for example, by the sound of modems. Today, it is silent and permanent. Data that used to be stored locally, in digital libraries or on physical media—hard drives, DVDs, CDs, etc.—are now stored on servers the user no longer owns, but consumes on demand via streaming over the Internet thanks to unlimited data plans. With each streaming session, this data is downloaded locally, used, and then silently discarded; each new use or rewind involves downloading it again and then discarding it again. These data does not belong to the user, who will no longer have access to it if they stop using the service or if the *cloud* service is no longer available.

The disposability of these data is a form of obsolescence that is particularly rendered invisible. Although single-use, online data have a material existence that is both multiple and disposable: the storage space they require on servers and data centers, but also on all personal computers where they pass through multiple cycles of copying and erasure. With each cycle of consumption and disposal, they consume remote, local, and infrastructural resources, which in turn require other resources for development, operation, and maintenance.

Also rendered invisible is the materiality of the cloud: data centers requiring large amounts of energy and water to operate, underground and undersea networks of copper cables or fiber optics, antennas for 2G, then 3G, 4G, and 5G standards, etc. The *wireless* connection is not wireless at all, it consists of intercontinental cables, wired networks, telecom antennas, and satellites [183].

Each of these infrastructures has its own vulnerabilities or exacerbates existing issues [24]. The rollout of 5G in 2020 accelerated the obsolescence of 2G and 3G networks and numerous connected devices (cars, elevators, phones, IoT devices) [25]. The same issue applies to investments in so-called AI systems, which require a major overhaul of data centers and their servers to accommodate new dedicated graphics chips and cooling systems.

Chips and systems do not necessarily replace the old ones but are added to them—we are clearly in a model of technological evolution through accumulation and hybridization, rather than through successive substitutions of one model for another, as described by Jean-Baptiste Fressoz [184].

Furthermore, these systems entail the launch of new products: smartwatches, smartphones incorporating dedicated AI chips, and 5G plans, touted by increasingly invasive communication tools that emphasize the 'modernity' of the infrastructure. Their obsolescence, to become socially acceptable, is presented as coextensive with innovation. This type of discourse is echoed at the highest levels of governments. In 2020, Emmanuel Macron announced that France would commit to 5G because *"it is the turning point of innovation"* [26]. In 2025, he declared at the AI Summit, *"when the world*

[24] Investigation: In Marseille as elsewhere, the takeover of territory by digital infrastructure, La Quadrature du Net, November 2024, last accessed in April 2026

[25] Why most countries are struggling to shut down 2G, restofworld.org, 7 March 2025, last accessed in April 2026.

[26] Speech by President Emmanuel Macron to representatives of the digital sector, elysee.fr, 14 Sept. 2020, last accessed in April 2026.

is accelerating, we cannot decide to slow down, we must go all out' [27]. Industrial investment is presented here as a natural evolution, even though there is no consensus on the necessity of such technology [185, 186].

4.4 Conclusion

Since the 1960s, there have been numerous cases of obsolescence caused by hardware and software. Simply listing these cases and describing their technical causes is not enough, given that the history of computing is so deeply embedded in narratives that portray obsolescence as a natural process. Furthermore, any regulation that treats obsolescence as a misdemeanor is countered by regulations that link it to intellectual property. It is therefore necessary to deconstruct this discourse. Rather than speaking of phenomena and causes of obsolescence, we will speak of strategies and opportunities for obsolescence. Through planning, communication, the de facto standardization of closed models, pseudo-history, and lobbying, obsolescence has been chosen and imposed, both as a strategy and as a narrative.

The history of computing is not merely a complex history of technological evolution. It encompasses a set of practices but also a history of constructing economic, legal, and rhetorical models aimed at planning production to artificially pace the release of products to market. But also aimed at enclosing software and digital systems, in order to secure ownership of innovation. And finally, aimed at producing discourses that present these maneuvers as the result of a vast law of nature. While many digital standards have become established de facto, 'de facto' does not mean 'by itself' without efforts to impose them: technical constraints and lobbying take over from legal constraints when resistance arises. Finally, it is by rendering invisible resistance, maintenance, care, hacking, repair, FOSS digital commons—all of which have been ever-present throughout the history of computing—that obsolescence has taken center stage at the heart of digital capitalism, as a business model.

[27] Emmanuel Macron : l'intelligence artificielle, "il faut y aller à fond", 7 Feb. 2025, Le Parisien, last accessed in April 2026.

Chapter 5

Key takeaways and conclusion

Contents

5.1 Key takeaways

In this doctoral dissertation I analyze software obsolescence and existing remediation strategies, in light of the significant environmental impact of digital technologies, particularly that of electronic devices. To that end, I study two software ecosystems, Android and Debian, by using a qualitative research approach.

5.1.1 Android findings and key takeaways

The first case study focuses on the Android ecosystem, the world's most widely used operating system (OS), based on a Linux kernel, where devices are often replaced and rarely updated more than two years after their release. I investigate what hinders Android development and maintenance. I conducted 12 interviews with key players in the ecosystem, supplemented by conference ethnography and analysis of technical literature.

The analysis shows that the way code flows are organized across the various Android ecosystem actors inhibits the updates, and outlines how these actors locate their maintenance efforts in different places to serve their strategic interests. The lack of updates appears in particular at the kernel level, i.e., at the core of Android builds, as the code from phone vendors and system on chip manufacturers increasingly diverges from the original Linux kernel code.

Reflecting on the Android findings, I notice the various strategies observed in terms of code flows between actors, and how they inhibit or facilitate maintenance. Maintenance breaking points

can be technical: silent break of software support, lack of updates and upgrades, code obfuscation, proprietary binary code, lack of documentation, hidden schematics, anti patterns in coding practices. This is particularly the case in the code provided by chip and smartphone vendors within the Android ecosystem. Google, the main actor governing the ecosystem, addresses maintenance issues by shifting responsibility towards phone vendors. However, as vendors are the least inclined actors to maintain their code, the problem persists, leading to premature end-of-life for devices and, consequently, their obsolescence.

Another cause of software obsolescence are the power dynamics within Android. Google, the primary driver of its development, plays a subtle game of opening and closing the code in the OS, and consolidates its dominance of the ecosystem in line with its economic interests. Maintenance of Android code by Google follows these interests. Although Google upstreams code to the Linux kernel to some extent, it does not mainline the Linux kernel. Moreover, its OS updates or upgardes often break retro-compatibility, and leave important pieces of the open Android code behind. At the same time, Google is focusing on key features within the ecosystem that have been developed as proprietary Google services and imposed on phone vendors through contractual means. This makes the entire ecosystem and its participants dependent on Google for essential services such as the Google Play Services.

In this ecosystem, driven by a concern for longevity, some vendors like Fairphone, Commown and some alternative free open source mobile actors such as LineageOS, Mobian, postmarketOS, implement remediation strategies to maintain devices and extend their lifetime. Based on their input, and the results of our study, I present a non-exhaustive list of socio-technical practices that help them offer OS updates even after SoC manufacturers, phone vendors or Google stop maintaining.

As a software development entity, looking to achieve long-term software support:

- mainline the Linux kernel;
- upstream code;
- reverse-engineer devices;
- share code and collaborate with other related FOSS projects;
- document code;
- share proprietary schematics with the FOSS communities;
- open source code;
- follow open development practices that foster collaboration;
- eliminate / minimize proprietary code that is not maintained, nor controlled by the community;
- eliminate / minimize blobs by patching;
- eliminate / minimize patches by upstreaming their code,
- openly communicate about work processes, release timelines, problems, needs;
- ask for user feedback when releasing new code;

- dedicate spaces to user feedback (forum, issue or bug trackers).

As a vendor looking to extend the lifetime of its devices:

- have a dedicated team for the OS and software updates;
- facilitate or offer the possibility to install alternative OSes on the device, especially when your OS support is finished;
- offer a dedicated software client support service, by phone or online forums;
- collaborate with the alternative software communities that offer open source support to your devices, especially when the chip manufacturer stops offering you software support;
- openly communicate your software support plans, timelines, and updates.

5.1.2 Debian findings and key takeaways

The second case study (chapter 3) focuses on maintenance within Debian, a widely spread free and open source operating system, also based on the Linux kernel, maintained by a large international volunteer community, following free open source code development practices and principles. I investigate maintenance in Debian, by use of a community ethnographic approach during gatherings. I conducted 11 semi-structured interviews, supplemented by informal discussions, participant observation, analysis of online community tools and technical literature. The data were analysed using thematic analysis.

I show that Debian's technical and social structure is designed to support and facilitate maintenance. This maintenance takes place at various levels: technical maintenance (code, tools), community maintenance (social and human interactions), work organisational maintenance (work processes, teams, roles) and infrastructuring (collective maintenance of inner infrastructure). At each of these levels, I show how maintenance in Debian is not only a technical process, but involves social interactions within and outside of the community.

I investigate, at each of these levels, what boosts maintenance in Debian, and what hinders it.

Maintenance boosts are located at both the socio-technical level of the packages, the work organizational one and at the financing one. Code in Debian is organized in packages. The analysis shows that the socio-technical relationship between Debian developers and external upstream developers plays an essential role in both the Debian package maintenance, and the upstream software maintenance.

I identify the crucial role that the collective process of developing, maintaining and improving the technical and social infrastructure in Debian plays in helping maintaining the project.

Our analysis identifies how The Long Term Support program in Debian—offering 5 years of support on each release—is an original business model that the community has established, providing it with the revenue needed for maintenance while limiting corporate influence on maintenance decisions.

Finally, the study identifies barriers in maintenance: human relations within the community: resolving conflicts, avoiding burnout, retaining members, attracting new ones seem to be important concerns for community members.

I now summarize some socio-technical practices that foster maintenance as observed in Debian.

At the software development level:

- mainline the Linux kernel;
- upstream as much code as possible;
- eliminate / minimize blobs and proprietary code;
- apply patches to eliminate blobs and proprietary code;
- minimize / eliminate patches by upstreaming;
- minimize package / software dependencies;
- develop and maintain good work relationship with the upstream developer community of your code;
- document your code; document your package; document your work;
- when developing or updating code, do not break retro-compatibility;
- if retro-compatibility is difficult to maintain, discuss the issues in teams and in the community of users or other developers to seek for solutions, and take decisions collectively;
- communicate about your work, achievements and issues with other members of the community of users or developers;
- ask for help or propose to help on community spaces.

As a community developing and maintaining software:

- develop and maintain the community hardware and software infrastructure;
- discuss issues within teams, between teams, with users, in community gatherings;
- develop work processes and document them;
- assign mentoring roles for new package maintainers;
- assign welcoming and support role for newcomers in the community;
- develop and maintain accessibility and diversity in the community and in your software;
- develop tools to automate work: package building tools, package tracking tools, testing tools;
- take important decisions collectively;
- port the OS in new devices architectures and share this experience;
- ask for user feedback and organize spaces where this feedback is possible;
- develop and maintain relations with other FOSS systems, developers or users;
- share your code / documentation in and outside of your community and organize spaces for contributions.

5.2 Obsolescence, a business model in digital capitalism

Expanding on the study on Android, I present several other hardware-related as well as software-related obsolescence strategies in connected devices and computers that illustrate how obsolescence opportunities are obstacles to extending the lifespan of digital devices.

In this study I also examine the prevailing view of the history of digital technology—where the rapid replacement of each model with a new one is seen as progress—and the definitions of obsolescence that underpin it. By drawing a connection between scholars' critical analysis of Moore's Law—which, rather than being a 'law', served as a roadmap for planing the production of chips and computers as a series of periodic updates, I argue that obsolescence follows the same path. Obsolescence in the digital realm guides production in a manner that is both deliberate—it is planned and these plans are published—and disguised—it is presented as a law of nature.

I argue that the privatization of software, through legal means such as Intellectual Property, patterns, restrictive licences, coupled with unfair business practices such as tie-in sales, misleading advertising, and market domination strategies, is a form of enclosure of software and operating systems. The study of Android also illustrates this point, where the Linux kernel—a digital common FOSS system [187]—serves as a basis to Google for enclosing Android, the most used operating system in the world. Obsolescence in the digital realm is made possible by business strategies and opportunities, through mutually reinforcing hardware and software techniques. Through planning, software lock-in, pseudo-history, and lobbying, obsolescence has been chosen and imposed, and lies at the heart of digital capitalism.

5.3 Recommendations

In the light of our findings, we present a non-exhaustive[1] list of recommendations concerning, on the one hand, coding practices and, on the other hand, some regulatory measures necessary to ensure software maintenance while prohibiting the abuse of dominant positions. Furthermore, it seems essential to implement public policies aimed at supporting and fostering fundamental software ecosystems such as operating systems.

[1] I welcome your suggestions for improvements.

Building on our analysis and insights from informants, we identified a set of practices that could support software maintenance:

- updates should not discard hardware;
- updates should not lead to software breaking changes and should provide retro-compatibility for long periods of time (e.g. min 10 years);
- if breaking changes were to happen, the updates should be set aside and discussed and evaluated with the user and developer communities affected by them until a common solution and process are established.

Because maintenance practices are enacted only when they align with the values and objectives of the stakeholders involved, it is also necessary to enforce them through regulation. The following recommendations range from the easiest measures to larger-scale transformations that would more radically improve longevity:

- require device vendors to publish update and upgrade plans for every device, and ensure that they are followed;
- require device vendors, chip and parts manufacturers as well as software companies to liberate all code, schematics and documentation related to hardware behaviour whenever they are no more providing updates, and upon expiration of the warranty and support period;
- include mainlining and upstreaming conditionality into Free and Open Source licences;
- require all operating system components, firmware and software in boot sequences to be maintained for at least 10 years;
- all operating system components, firmware and software in boot sequences should be required to follow (or define) open public standards, to avoid proprietary software lock-ins;
- consider operating systems, firmware and software in boot sequences as public infrastructure, governed as digital commons.

5.4 Conclusion

By examining the Android and Debian ecosystems, this research has identified and analysed important maintenance breaking points in software causing obsolescence. These breaking points can be technical: silent break of software support, lack of updates and upgrades, code obfuscation, proprietary and binary code, lack of documentation, hidden schematics, anti patterns in coding practices. This is for example the case in the code provided by chip and smartphone vendors within the Android ecosystem. Maintenance breaking points can also be socio-economic, supported by technical and legal instruments, creating dynamics of power and abuse of power in software ecosystems. This is particularly true of Google in the Android ecosystem.

The findings highlight the importance of free and open source software development practices in maintaining software systems on the long term, and extending the lifespan of devices. The study identifies coding practices that are important maintenance strategies at the software development level. But it is the collective act of maintaining, social interactions within the communities developing the software and with external communities of users or developers, that are key to long term maintenance.

As maintenance practices in Debian show, software maintenance is not limited to applying code updates, but it is a socio-technical process that involves discussing, analyzing, and preparing these updates so that they align with the needs of the user and developer communities affected by them, without disruptions.

Obsolescence—even though it is presented as such—is not inevitable. As the example of Debian and free and open-source mobile systems analysed in this study show, developing software for long-lasting devices is possible and strategies to achieve it are collectively discussed, implemented, and improved in these alternative spaces.

Bibliography

[1] Maud Lecarpentier and Elsa Lecarpentier. Toujours puce: les macrodégâts de la microélectronique. Le monde à l'envers, 2024. ISBN 979-10-91772-58-7.

[2] Johan Rockström, Will Steffen, Kevin Noone, Åsa Persson, F. Chapin, Eric Lambin, Timothy Lenton, Marten Scheffer, Carl Folke, Schellnhuber, Björn Nykvist, Cynthia Wit, Terry Hughes, Sander Leeuw, Henning Rodhe, Sverker Sörlin, Peter Snyder, Robert Costanza, Uno Svedin, Malin Falkenmark, Louise Karlberg, Robert Corell, Victoria Fabry, James Hansen, Brian Walker, Diana Liverman, Katherine Richardson, Paul Crutzen, and Jonathan Foley. Ecology and Society: Planetary Boundaries: Exploring the Safe Operating Space for Humanity. In Ecology and Society, volume 14, November 2009. doi: 10.5751/ES-03180-140232.

[3] Will Steffen. Introducing the Anthropocene: The human epoch. In Ambio, volume 50, pages 1784–1787, October 2021. doi: 10.1007/s13280-020-01489-4.

[4] Johan Rockström, Joyeeta Gupta, Dahe Qin, Steven Lade, Jesse Abrams, Lauren Andersen, David Armstrong McKay, Xuemei Bai, Govindasamy Bala, Stuart E. Bunn, Daniel Ciobanu, Fabrice DeClerck, Kristie Ebi, Lauren Gifford, Christopher Gordon, Syezlin Hasan, Norichika Kanie, Timothy Lenton, Sina Loriani, Diana Liverman, Awaz Mohamed, Nebojsa Nakicenovic, David Obura, Daniel Ospina, Klaudia Prodani, Crelis Rammelt, Boris Sakschewski, Joeri Scholtens, Ben Stewart-Koster, Thejna Tharammal, Detlef Vuuren, Peter Verburg, Ricarda Winkelmann, Caroline Zimm, Elena Bennett, Stefan Bringezu, Wendy Broadgate, Pamela Green, Lei Huang, Lisa Jacobson, Christopher Ndehedehe, Simona Pedde, Juan Rocha, Marten Scheffer, Lena Schulte-Uebbing, Wim Vries, Cunde Xiao, Chi Xu, Xinwu Xu, Noelia Zafra-Calvo, and Xin Zhang. Safe and just Earth system boundaries. In Nature, volume 619, pages 102–111. Nature Publishing Group, July 2023. doi: 10.1038/s41586-023-06083-8.

[5] Leena Heinämäki. Human Rights and the Environment. Yearbook of International Environmental Law, 33(1):29–36, January 2022. ISSN 2045-0052. doi: 10.1093/yiel/yvad005.

[6] Neil J. W. Crawford, Kavya Michael, and Michael Mikulewicz, editors. Climate Justice in the Majority World: Vulnerability, Resistance, and Diverse Knowledges. Routledge, London, July 2023. ISBN 978-1-003-21402-1. doi: 10.4324/9781003214021.

[7] Dana R. Fisher and Sohana Nasrin. Climate activism and its effects. In WIREs Climate Change, volume 12, page e683, 2021. doi: 10.1002/wcc.683.

[8] Hugues Ferreboeuf, Zeynep Kahraman, Maxime Efoui-Hess, Françoise Berthoud, Philippe Bihouix, Pierre Fabre, Daniel Kaplan, Laurent Lefèvre, Alexandre Monnin, Olivier Ridoux,

Samuli Vaija, Marc Vautier, Xavier Verne, and Alain Ducass. Lean ICT: Towards digital sobriety. Technical report, The Shift Project, 2019.

[9] Charlotte Freitag, Mike Berners-Lee, Kelly Widdicks, Bran Knowles, Gordon S. Blair, and Adrian Friday. The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations. In Patterns, volume 2, page 100340, September 2021. doi: 10.1016/j.patter.2021.100340.

[10] Gereon Mewes. The Digital Environmental Footprint - a holistic framework of Digital Sustainability. EarthArXiv, March 2023.

[11] Anne Pasek, Hunter Vaughan, and Nicole Starosielski. The world wide web of carbon: Toward a relational footprinting of information and communications technology's climate impacts. In Big Data & Society, volume 10, page 20539517231158994. SAGE Publications Ltd, January 2023. doi: 10.1177/20539517231158994.

[12] Françoise Berthoud, Pascal Guitton, Laurent Lefèvre, Sophie Quinton, Antoine Rousseau, Jacques Sainte-Marie, Céline Serrano, Jean-Bernard Stefani, Peter Sturm, and Eric Tannier. Sciences, Environnements et Sociétés. Other, Inria, October 2019.

[13] Kelly Widdicks, Federica Lucivero, Gabrielle Samuel, Lucas Somavilla Croxatto, Marcia Tavares Smith, Carolyn Ten Holter, Mike Berners-Lee, Gordon S. Blair, Marina Jirotka, Bran Knowles, Steven Sorrell, Miriam Börjesson Rivera, Caroline Cook, Vlad C. Coroamă, Timothy J. Foxon, Jeffrey Hardy, Lorenz M. Hilty, Simon Hinterholzer, and Birgit Penzenstadler. Systems thinking and efficiency under emissions constraints: Addressing rebound effects in digital innovation and policy. In Patterns (New York, N.Y.), volume 4, page 100679, February 2023. doi: 10.1016/j.patter.2023.100679.

[14] Gauthier Roussilhe, Anne-Laure Ligozat, and Sophie Quinton. A long road ahead: A review of the state of knowledge of the environmental effects of digitization. In Current Opinion in Environmental Sustainability, volume 62, page 101296, June 2023. doi: 10.1016/j.cosust.2023.101296.

[15] Ana Valdivia. The supply chain capitalism of AI: A call to (re)think algorithmic harms and resistance through environmental lens. In Information, Communication & Society, volume 28, pages 2118–2134. Routledge, September 2025. doi: 10.1080/1369118X.2024.2420021.

[16] Eric Tannier, Vincent Daubin, and Sophie Quinton. The crisis of the scientific mind : An investigation, a tragedy and a collective redistribution of roles. In Les Cahiers de Framespa : E-Storia, number 40. laboratoire FRAMESPA (UMR 5136, Université Toulouse - Jean Jaurès / CNRS), June 2022. doi: 10.4000/framespa.13150.

[17] Florence Maraninchi. Let us not put all our eggs in one basket: Towards New Research Directions in Computer Science. In Communications of the ACM, volume 65, pages 35–37, August 2022. doi: 10.1145/3528088.

[18] Florence Maraninchi. Revisiting "Good" Software Design Principles To Shape Undone Computer Science Topics. In Undone Computer Science Conference, Nantes, France, February 2024. Université de Nantes and Inria and LORIA Université de Lorraine.

[19] Scott Frickel, Sahra Gibbon, Jeff Howard, Joanna Kempner, Gwen Ottinger, and David J. Hess. Undone Science: Charting Social Movement and Civil Society Challenges to Research Agenda Setting. In Science, Technology, & Human Values, volume 35, pages 444–473. SAGE Publications Inc, July 2010. doi: 10.1177/0162243909345836.

[20] David J. Hess. Undone Science: Social Movements, Mobilized Publics, and Industrial Transitions. The MIT Press, October 2016. ISBN 978-0-262-03513-2. doi: 10.7551/mitpress/9780262035132.001.0001.

[21] Aurélie Bugeau and Anne-Laure Ligozat. Analysing ICT in prospective scenarios to help reveal undone computer science. In Undone Computer Science, Nantes, France, February 2024.

[22] Felienne Hermans. The Computer Science Undone: How The Social Construction of Disciplinary Boundaries and Disciplinary Hierarchies Shape a Field. In Philosophia Scientiæ, volume 30, pages 3–30, 2026.

[23] Will Steffen, Katherine Richardson, Johan Rockström, Sarah Cornell, Ingo Fetzer, Elena Bennett, Reinette Biggs, Stephen Carpenter, W Vries, Cynthia Wit, Carl Folke, Dieter Gerten, Jens Heinke, Georgina Mace, Linn Persson, Veerabhadran Ramanathan, Belinda Reyers, and Sverker Sörlin. Planetary boundaries: Guiding human development on a changing planet. In Science, volume 347. American Association for the Advancement of Science, February 2015. doi: 10.1126/science.1259855.

[24] S. Lange, T. Santarius, L. Dencik, T. Diez, H. Ferreboeuf, S. Hankey, A. Hilbeck, L. Hilly, M. Hoejer, D. Kleine, J. Pohl, L. Reisch, M. Ryghaug, T. Schwanen, and P. Staab. Digital reset: Redirecting technologies for the deep sustainability transformation. Technische Universitaet Berlin, 2022. doi: 10.14279/depositonce-16187.

[25] T. Santarius, L. Dencik, T. Diez, H. Ferreboeuf, P. Jankowski, S. Hankey, A. Hilbeck, L. M. Hilty, M. Höjer, D. Kleine, S. Lange, J. Pohl, L. Reisch, M. Ryghaug, T. Schwanen, and P. Staab. Digitalization and Sustainability: A Call for a Digital Green Deal. In Environmental Science & Policy, volume 147, pages 11–14, September 2023. doi: 10.1016/j.envsci.2023.04.020.

[26] Céline Péréa, Jessica Gérard, and Julien Benedittis. Digital sobriety: From awareness of the negative impacts of IT usages to degrowth technology at work. In Technological Forecasting and Social Change, volume 194, page 122670, September 2023. doi: 10.1016/j.techfore.2023.122670.

[27] Charlotte Freitag, Mike Berners-Lee, Kelly Widdicks, Bran Knowles, Gordon S. Blair, and Adrian Friday. The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations. In Patterns, volume 2, page 100340, September 2021. doi: 10.1016/j.patter.2021.100340.

[28] Thomas Brilland, Erwann Fangeat, Julia Meyer, and Mathieu Wellhoff. Evaluation de l'impact environnemental du numérique en France. page 35. ADEME-Arcep, 2025.

[29] Jeremy Bourgoin, Roberto Interdonato, Quentin Grislain, Matteo Zignani, and Sabrina Gaito. Mining resources, the inconvenient truth of the "ecological" transition. In World

Development Perspectives, volume 35, page 100615, September 2024. doi: 10.1016/j.wdp.2024.100615.

[30] Eric D. Williams, Robert U. Ayres, and Miriam Heller. The 1.7 Kilogram Microchip: Energy and Material Use in the Production of Semiconductor Devices. In Environmental Science & Technology, volume 36, pages 5504–5510. American Chemical Society, December 2002. doi: 10.1021/es025643o.

[31] Christoph N. Vogel. Conflict Minerals, Inc.: War, Profit and White Saviourism in Eastern Congo. Oxford University Press, July 2022. ISBN 978-0-19-767649-3.

[32] Oche Joseph Otorkpa, Joseph Omeiza Alao, and Stephen Emmanuel. Environmental and Health Impacts of Unregulated Lithium Mining Practices: Lessons from Nigeria's Oil Industry. In African Journal of Environment and Natural Science Research, volume 7, pages 1–4, July 2024. doi: 10.52589/ajensr-h1og8f5u.

[33] Melissa Gregg. In Chips We Trust?: Remembering the Environmental Impacts of Hardware Manufacturing. In Centre for Media, Technology and Democracy Climate Justice & Technology Series. McGill University, April 2024.

[34] Thiane Neves-Barros. Reflections on the impact of digital infrastructure and racism in traditional communities. Association for Progressive Communication, November 2025.

[35] Toussaint Nothias. An intellectual history of digital colonialism. In Journal of Communication, volume 75, pages 385–397, October 2025. doi: 10.1093/joc/jqaf003.

[36] Fieke Jansen, Merve Gülmez, Becky Kazansky, Narmine Abou Bakari, Claire Fernandez, Harriet Kingaby, and Jan Tobias Mühlberg. The Climate Crisis is a Digital Rights Crisis: Exploring the Civil-Society Framing of Two Intersecting Disasters. In Computing within Limits. LIMITS, June 2023. doi: 10.21428/bf6fb269.b4704652.

[37] Marion Ficher, Tom Bauer, and Anne-Laure Ligozat. A comprehensive review of the end-of-life modeling in LCAs of digital equipment. In The International Journal of Life Cycle Assessment, volume 30, pages 20–42, January 2025. doi: 10.1007/s11367-024-02367-x.

[38] Josh Lepawsky. Reassembling Rubbish: Worlding Electronic Waste. MIT Press, 2018.

[39] Michael Braungart and William McDonough. Cradle to Cradle. Random House, January 2009. ISBN 978-1-4070-2132-4.

[40] Marcel Den Hollander, C.A. Bakker, and Erik Hultink. Product Design in a Circular Economy: Development of a Typology of Key Concepts and Terms: Key Concepts and Terms for Circular Product Design. In Journal of Industrial Ecology, volume 21, May 2017. doi: 10.1111/jiec.12610.

[41] Marcel Den Hollander. Design for Managing Obsolescence: A Design Methodology for Preserving Product Integrity in a Circular Economy. 2018. ISBN 978-90-828736-0-3. doi: 10.4233/uuid:3f2b2c52-7774-4384-a2fd-7201688237af.

[42] Jukka Manner. Black software — the energy unsustainability of software systems in the 21st century. In Oxford Open Energy, volume 2, page oiac011, February 2023. doi: 10.1093/ooenergy/oiac011.

[43] Christoph Schneider and Stefanie Betz. Transformation^2: Making software engineering accountable for sustainability. In Journal of Responsible Technology, volume 10, page 100027, July 2022. doi: 10.1016/j.jrt.2022.100027.

[44] Stefanie Betz, Birgit Penzenstadler, Leticia Duboc, Ruzanna Chitchyan, Sedef Akinli Kocak, Ian Brooks, Shola Oyedeji, Jari Porras, Norbert Seyff, and Colin C. Venters. Lessons Learned from Developing a Sustainability Awareness Framework for Software Engineering Using Design Science. In ACM Trans. Softw. Eng. Methodol., volume 33, pages 136:1–136:39. ACM, June 2024. doi: 10.1145/3649597.

[45] Nina Troeger, Harald Wieser, and Renate Hübner. Smartphones Are Replaced More Frequantly than T-Shirts: Patterns of Consumer Use and Reasons for Replacing Durable Goods. Arbeiterkammer of Austria, Vienna, February 2017. ISBN 978-3-7063-0669-0.

[46] Lise Magnier and Ruth Mugge. Replaced too soon? An exploration of Western European consumers' replacement of electronic products. In Resources, Conservation and Recycling, volume 185, page 106448, October 2022. doi: 10.1016/j.resconrec.2022.106448.

[47] Dmitry Zhilyaev, Ciprian Cimpan, Zhi Cao, Gang Liu, Søren Askegaard, and Henrik Wenzel. The living, the dead, and the obsolete: A characterization of lifetime and stock of ICT products in Denmark. In Resources, Conservation and Recycling, volume 164, page 105117, January 2021. doi: 10.1016/j.resconrec.2020.105117.

[48] Shailesh Prabhu N and Ritanjali Majhi. Disposal of obsolete mobile phones: A review on replacement, disposal methods, in-use lifespan, reuse and recycling. In Waste Management & Research, volume 41, pages 18–36. SAGE Publications Ltd STM, January 2023. doi: 10.1177/0734242X221105429.

[49] Edlira Nano and Jeanne Guien. L'obsolescence : modèle économique du capitalisme numérique. In Capitalisme numérique. C&f edition.

[50] Marina Proske and Melanie Jaeger-Erben. Decreasing obsolescence with modular smartphones? – An interdisciplinary perspective on lifecycles. In Journal of Cleaner Production, volume 223, pages 57–66, June 2019. doi: 10.1016/j.jclepro.2019.03.116.

[51] Michael Pecht, Rajeev Solomon, Peter Sandborn, Chris Wilkinson, and Diganta Das. Obsolescence Prediction and Management. In Parts Selection and Management, pages 231–263. John Wiley & Sons, Ltd, 2004. ISBN 978-0-471-72388-2. doi: 10.1002/0471723886.ch16.

[52] L. Merola. The COTS software obsolescence threat. In Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'05), pages 7 pp.–, February 2006. doi: 10.1109/ICCBSS.2006.29.

[53] Peter A. Sandborn, Frank Mauro, and Ron Knox. A Data Mining Based Approach to Electronic Part Obsolescence Forecasting. In IEEE Transactions on Components and

Packaging Technologies, volume 30, pages 397–401, September 2007. doi: 10.1109/TCAPT.2007.900058.

[54] Peter Sandborn. Software Obsolescence: Complicating the Part and Technology Obsolescence Management Problem. In Components and Packaging Technologies, IEEE Transactions On, volume 30, pages 886–888, January 2008. doi: 10.1109/TCAPT.2007.910918.

[55] Bill Gates and Collins Hemingway. Business @ the Speed of Thought: Using a Digital Nervous System. Penguin, 1999. ISBN 978-0-14-028311-2.

[56] Kiri Feldman and Peter Sandborn. Integrating Technology Obsolescence Considerations Into Product Design Planning. In ASME 2007 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, pages 981–988. American Society of Mechanical Engineers Digital Collection, May 2009. doi: 10.1115/DETC2007-35881.

[57] Peter Sandborn. Design for Obsolescence Risk Management. In Procedia CIRP, volume 11 of 2nd International Through-life Engineering Services Conference, pages 15–22, January 2013. doi: 10.1016/j.procir.2013.07.073.

[58] Bartels, Ermel, Sandborn, and Pecht. Software Obsolescence. In Strategies to the Prediction, Mitigation and Management of Product Obsolescence, chapter 6, pages 143–155. John Wiley & Sons, Ltd, 2012. ISBN 978-1-118-27547-4. doi: 10.1002/9781118275474.ch6.

[59] Jeanne Guien. Le consumérisme à travers ses objets. Editions Divergences, 2021.

[60] Jeanne Guien. Une histoire des produits menstruels. Éditions Divergences, 2023. ISBN 979-10-97088-55-2.

[61] Susan Strasser. Waste and Want : A Social History of Trash. New York, N.Y. : Henry Holt and Co., 2000. ISBN 978-0-8050-6512-1.

[62] Marina Proske, Janis Winzer, Max Marwede, Nils F. Nissen, and Klaus-Dieter Lang. Obsolescence of electronics - the example of smartphones. In 2016 Electronics Goes Green 2016+ (EGG), pages 1–8, September 2016. doi: 10.1109/EGG.2016.7829852.

[63] Giles Slade. Made to Break: Technology and Obsolescence in America. Harvard University Press, 2006. ISBN 978-0-674-02572-1.

[64] Jeremy Bulow. An Economic Theory of Planned Obsolescence. In The Quarterly Journal of Economics, volume 101, pages 729–749. Oxford University Press, 1986. doi: 10.2307/1884176.

[65] Barak Orbach. The Durapolist Puzzle: Monopoly Power in Durable-Goods Market. Rochester, NY, June 2007. Social Science Research Network.

[66] Vance Packard. The Waste Makers. David McKay, 1960. ISBN 978-1-935439-37-0.

[67] Esra Karakuş Umar and Rafet Beyaz. Planned Obsolescence: Is It a Trap Set for the Consumer or Is It a Strategy Contributing to Social Development? In Ege Academic Review, volume 21, pages 181–191. Ege University, June 2021. doi: 10.21121/eab.953538.

[68] Michael Cox and Richard Alm. Creative Destruction. In The Concise Encylopedia of Economics.

[69] Nigel Whiteley. Toward a Throw-Away Culture. Consumerism, 'Style Obsolescence' and Cultural Theory in the 1950s and 1960s. In Oxford Art Journal, volume 10, pages 3–27. Oxford University Press, 1987.

[70] F Michel, T Huppertz, J. R. Dulbecco, and J Lhotellier. Evaluation économique de l'allongement de la durée d'usage de produits de consommation et biens d'équipement. page 149. ADEME, December 2019.

[71] Hubblo, CNRS, INRIA, and ADEME. Analyse conséquentielle des leviers d'écoconception des services numériques. Technical report, ADEME, April 2026.

[72] Jeanne Guien. Le désir de nouveautés: L'obsolescence au cœur du capitalisme (XVe-XXIe siècle). La Découverte, March 2025. ISBN 978-2-348-08343-3.

[73] Mário Barros and Eric Dimla. From Planned Obsolescence to the Circular Economy in the Smartphone Industry: An Evolution of Strategies Embodied in Product Features. In Design Society, 2021. doi: 10.1017/pds.2021.422.

[74] Giorgio Di Natale, Elena Ioana Vatajelu, Francesco Regazzoni, Betina Zynger-Capaverde, and Alua Ibraim. Design for Sustainability: A New Philosophy for Addressing Chip Shortage. March 2026.

[75] Ludmila Courtillat-Piazza, Sophie Quinton, and Clement Marquet. Digitalisation as threat to resilience: What if there are no more semiconductors? May 2024.

[76] Esther Jang, Matthew Johnson, Edward Burnell, and Kurtis Heimerl. Unplanned Obsolescence: Hardware and Software After Collapse. In Proceedings of the 2017 Workshop on Computing Within Limits, LIMITS '17, pages 93–101, New York, NY, USA, June 2017. Association for Computing Machinery. ISBN 978-1-4503-4950-5. doi: 10.1145/3080556.3080566.

[77] Kênia Pereira Batista Webster, Káthia Marçal De Oliveira, and Nicolas Anquetil. A risk taxonomy proposal for software maintenance. In 21st IEEE International Conference on Software Maintenance (ICSM'05), pages 453–461. IEEE, 2005. doi: 10.1109/ICSM.2005.14.

[78] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. In IEEE Transactions on Software Engineering, volume 27 of IEEE Transactions on Software Engineering, pages 1–12, January 2001. doi: 10.1109/32.895984.

[79] Gerardo Canfora and Aniello Cimitile. Software maintenance. In Handbook of Software Engineering and Knowledge Engineering, pages 91–120. World Scientific Publishing Company, December 2001. ISBN 978-981-02-4973-1. doi: 10.1142/9789812389718_0005.

[80] Harald Wieser and Nina Tröger. Exploring the inner loops of the circular economy: Replacement, repair, and reuse of mobile phones in Austria. In Journal of Cleaner Production, volume 172, pages 3042–3055, January 2018. doi: 10.1016/j.jclepro.2017.11.106.

[81] Francesco Vitale, Joanna McGrenere, Aurélien Tabard, Michel Beaudouin-Lafon, and Wendy E. Mackay. High Costs and Small Benefits: A Field Study of How Users Experience Operating System Upgrades. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, CHI '17, pages 4242–4253, New York, NY, USA, May 2017. Association for Computing Machinery. ISBN 978-1-4503-4655-9. doi: 10.1145/3025453.3025509.

[82] R. Stuart Geiger, Dorothy Howard, and Lilly Irani. The Labor of Maintaining and Scaling Free and Open-Source Software Projects. In Proc. ACM Hum.-Comput. Interact., volume 5, pages 175:1–175:28, April 2021. doi: 10.1145/3449249.

[83] Janet Vertesi and J. Nathan Matias. Divesting from Big Tech: Alternative Possibilities for Research and Futuring in Social Computing. In Companion Publication of the 2023 Conference on Computer Supported Cooperative Work and Social Computing, CSCW '23 Companion, pages 401–404, New York, NY, USA, October 2023. Association for Computing Machinery. ISBN 979-8-4007-0129-0. doi: 10.1145/3584931.3608436.

[84] Stephen Graham and Nigel Thrift. Out of Order: Understanding Repair and Maintenance. In Theory, Culture & Society, volume 24, pages 1–25. SAGE Publications Ltd, May 2007. doi: 10.1177/0263276407075954.

[85] Jérôme Denis, Alessandro Mongili, and David Pontille. Maintenance & Repair in Science and Technology Studies. In Tecnoscienza – Italian Journal of Science & Technology Studies, volume 6, pages 5–15, 2015. doi: 10.6092/issn.2038-3460/17251.

[86] Jérôme Denis and David Pontille. Material Ordering and the Care of Things. In Science, Technology, & Human Values, volume 40, pages 338–367. SAGE Publications Inc, May 2015. doi: 10.1177/0162243914553129.

[87] Annemarie Mol. The Logic of Care: Health and the Problem of Patient Choice. Routledge, London, 1st edition, 2008. doi: 10.4324/9780203927076.

[88] Lesly Sierra-Fontalvo, Arturo Gonzalez-Quiroga, and Jaime A. Mesa. A deep dive into addressing obsolescence in product design: A review. In Heliyon, volume 9, page e21856, November 2023. doi: 10.1016/j.heliyon.2023.e21856.

[89] Marisa Leavitt Cohn. Convivial Decay: Entangled Lifetimes in a Geriatric Infrastructure. In Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing, CSCW '16, pages 1511–1523, New York, NY, USA, February 2016. Association for Computing Machinery. ISBN 978-1-4503-3592-8. doi: 10.1145/2818048.2820077.

[90] Daniela K. Rosner and Morgan Ames. Designing for repair? infrastructures and materialities of breakdown. In Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing, CSCW '14, pages 319–331, New York, NY, USA, February 2014. Association for Computing Machinery. ISBN 978-1-4503-2540-0. doi: 10.1145/2531602.2531692.

[91] Steven J. Jackson and Laewoo Kang. Breakdown, obsolescence and reuse: HCI and the art of repair. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,

CHI '14, pages 449–458, New York, NY, USA, April 2014. Association for Computing Machinery. ISBN 978-1-4503-2473-1. doi: 10.1145/2556288.2557332.

[92] Laura Kocksch and Torben Elgaard Jensen. The Mundane Art of Cybersecurity: Living with Insecure IT in Danish Small- and Medium-Sized Enterprises. In Proc. ACM Hum.-Comput. Interact., volume 8, pages 354:1–354:17, November 2024. doi: 10.1145/3686893.

[93] Irene Maldini, Ingun Grimstad Klepp, and Kirsi Laitala. The environmental impact of product lifetime extension: A literature review and research agenda. In Sustainable Production and Consumption, volume 56, pages 561–578, June 2025. doi: 10.1016/j.spc.2025.04.020.

[94] Léa Mosesso, Nolwenn Maudet, Edlira Nano, Thomas Thibault, and Aurélien Tabard. Obsolescence Paths: Living with aging devices. In ICT4S 2023 - International Conference on Information and Communications Technology for Sustainability, Rennes, France, June 2023. doi: 10.1109/ICT4S58814.2023.00011.

[95] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pages 226–237, Singapore Singapore, August 2016. ACM. ISBN 978-1-4503-3845-5. doi: 10.1145/2970276.2970312.

[96] Ben Fiedler, Daniel Schwyn, Constantin Gierczak-Galle, David Cock, and Timothy Roscoe. Putting out the hardware dumpster fire. In Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS '23, pages 46–52, New York, NY, USA, June 2023. Association for Computing Machinery. ISBN 979-8-4007-0195-5. doi: 10.1145/3593856.3595903.

[97] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs. In 2012 19th Working Conference on Reverse Engineering, pages 83–92, October 2012. doi: 10.1109/WCRE.2012.18.

[98] Muhammad Kamran, Junaid Rashid, and Muhammad Nisar. Android Fragmentation Classification, Causes, Problems and Solutions. In International Journal of Computer Network and Information Security, volume 14, pages 992–999, September 2016.

[99] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. API change and fault proneness: A threat to the success of Android apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pages 477–487, New York, NY, USA, August 2013. Association for Computing Machinery. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491428.

[100] Sherlock A. Licorish, Amjed Tahir, Michael Franklin Bosu, and Stephen G. MacDonell. On satisfying the Android OS community: User feedback still central to developers' portfolios. In 2015 24th Australasian Software Engineering Conference, pages 78–87, Adelaide, SA, Australia, 2015. IEEE. doi: 10.1109/ASWEC.2015.19.

[101] Daniel R. Thomas, Alastair R. Beresford, and Andrew Rice. Security Metrics for the Android Ecosystem. In Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM '15, pages 87–98, New York, NY, USA, October 2015. Association for Computing Machinery. ISBN 978-1-4503-3819-6. doi: 10.1145/2808117.2808118.

[102] Ernst Leierzopf, René Mayrhofer, Michael Roland, Wolfgang Studier, Lawrence Dean, Martin Seiffert, Florentin Putz, Lucas Becker, and Daniel R. Thomas. A Data-Driven Evaluation of the Current Security State of Android Devices. In 2024 IEEE Conference on Communications and Network Security (CNS), pages 1–9, September 2024. doi: 10.1109/CNS62487.2024.10735682.

[103] Matthias Korn and Susann Wagenknecht. Friction in Arenas of Repair: Hacking, Security Research, and Mobile Phone Infrastructure. In Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing, CSCW '17, pages 2475–2488, New York, NY, USA, February 2017. Association for Computing Machinery. ISBN 978-1-4503-4335-0. doi: 10.1145/2998181.2998308.

[104] Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly, and Lee Todd. Surviving the SOC Revolution. Kluwer Academic Publishers, New York, 2002. ISBN 978-0-7923-8679-7. doi: 10.1007/b116290.

[105] Ahmed Amine Jerraya, Sungjoo Yoo, Norbert Wehn, and Diederik Verkest. Embedded Software for SoC. Springer Publishing Company, Incorporated, September 2013. ISBN 978-1-4757-8499-2.

[106] Samuel Greengard. Fighting for the Right to Repair. In Communications of The Acm, September 2025.

[107] Right to Repair coalition and iFixit. The Current State of Right to Repair in the EU: A Snapshot. The Right To Repair Europe Coalition, January 2025.

[108] Sahra Svensson, Jessika Luth Richter, Eléonore Maitre-Ekern, Taina Pihlajarinne, Aline Maigret, and Carl Dalhammar. The Emerging 'Right to Repair' legislation in the EU and the U.S. In Going Green CARE INNOVATION 2018, 2018.

[109] Javier Lloveras, Mario Pansera, and Adrian Smith. On 'the Politics of Repair Beyond Repair': Radical Democracy and the Right to Repair Movement. In Journal of Business Ethics, volume 196, pages 325–344, January 2025. doi: 10.1007/s10551-024-05705-z.

[110] Dunia P. Zongwe. The Economics of Repair: Fixing Planned Obsolescence by Activating the Right to Repair in India. In International Journal on Consumer Law and Practice, volume 11, January 2023.

[111] Lachlan D Urquhart, Susan Lechelt, Christopher Boniface, Haili Wu, Anna Marie Rezk, Nidhi Dubey, Melissa Terras, and Ewa Luger. The Right to Repair (R2R) Cards: Aligning Law and Design For A More Sustainable Consumer Internet of Things. In Proceedings of the 13th Nordic Conference on Human-Computer Interaction, NordiCHI '24, pages 1–20, New York, NY, USA, October 2024. Association for Computing Machinery. ISBN 979-8-4007-0966-1. doi: 10.1145/3679318.3685341.

[112] Sarah Suzaña. The European Commission vs. Google: Analysis of the cases AT. 40099 (Google Android) and AT. 40411 (Google AdSense) for Abuse of Dominant Position. In Università Della Calabria, 2021.

[113] Federico Etro and Cristina Caffarra. On the economics of the Android case. In European Competition Journal, volume 13, pages 282–313. Routledge, September 2017. doi: 10.1080/17441056.2017.1386957.

[114] C. Paul Rogers. Competition Law and the E.U. and U.S. Approaches to Dominant Markets: Will the Gap Narrow? Rochester, NY, 2021. Social Science Research Network.

[115] Competition and Markets Authority of the UK Government. Google's agreements with device manufacturers and app developers. Technical Report Appendix E, June 2022.

[116] Competition and Markets Authority Of The Uk Government. Mobile ecosystems market study. Technical report, June 2022.

[117] Andrew S. Tanenbaum and Herbert Bos. Modern Operating Systems. Pearson Education, global edition, 2015. ISBN 978-0-13-359162-0 978-1-292-06142-9.

[118] Cristiano Politowski, Foutse Khomh, Simone Romano, Giuseppe Scanniello, Fabio Petrillo, Yann-Gaël Guéhéneuc, and Abdou Maiga. A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension. In Information and Software Technology, volume 122, page 106278, June 2020. doi: 10.1016/j.infsof.2020.106278.

[119] Xuetao Li, Yuxia Zhang, Cailean Osborne, Minghui Zhou, Zhi Jin, and Hui Liu. Systematic Literature Review of Commercial Participation in Open Source Software. In ACM Trans. Softw. Eng. Methodol., volume 34, pages 33:1–33:31, January 2025. doi: 10.1145/3690632.

[120] Daniel Feitosa, Apostolos Ampatzoglou, Antonios Gkortzis, Stamatia Bibi, and Alexander Chatzigeorgiou. CODE reuse in practice: Benefiting or harming technical debt. In Journal of Systems and Software, volume 167, page 110618, September 2020. doi: 10.1016/j.jss.2020.110618.

[121] Ibrahim Haddad and Cedric Bail. Technical Debt and Open Source Development. 2020.

[122] The Linux Kernel Media team. Report of the Kernel CAM topic. Minutes of two days of meeting., The Linux Kernel Media Mini Summit, Dublin, September 2022.

[123] Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In 2011 15th European Conference on Software Maintenance and Reengineering, pages 181–190, March 2011. doi: 10.1109/CSMR.2011.24.

[124] A. Von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. In Computer, volume 28, pages 44–55, August 1995. doi: 10.1109/2.402076.

[125] Ruven Brooks. Towards a theory of the comprehension of computer programs. In International Journal of Man-Machine Studies, volume 18, pages 543–554, June 1983. doi: 10.1016/S0020-7373(83)80031-5.

[126] M.M. Lehman. Programs, life cycles, and laws of software evolution. In Proceedings of the IEEE, volume 68, pages 1060–1076, September 1980. doi: 10.1109/PROC.1980.11805.

[127] Frédéric Marty. Pré-installations, biais de statu quo et consolidation de la dominance : Les enseignements de l'arrêt du Tribunal de l'U.E. dans l'affaire Google Android. In CIRANO, volume 2022s-29, CIRANO, November 2022. doi: 10.54932/YOZL1587.

[128] Nidhi Modi and Trusha Modi. A Cross-Comparative Analysis of the Google's Self Preferencing and Android Cases in India, EU and the US. In Competition and Regulation in Network Industries, volume 25, pages 174–191. SAGE Publications Ltd STM, December 2024. doi: 10.1177/17835917251392879.

[129] Nadia Eghbal. Working in Public: The Making and Maintenance of Open Source Software. Stripe Press, August 2020. ISBN 978-1-953953-30-8.

[130] Hamid R. Ekbia and Bonnie A. Nardi. Heteromation, and Other Stories of Computing and Capitalism. The MIT Press, May 2017. ISBN 978-0-262-34032-8. doi: 10.7551/mitpress/10767.001.0001.

[131] Martin F. Krafft. The Debian System: Concepts and Techniques. No Starch Press, 2005. ISBN 978-1-59327-069-8.

[132] Jesus M. Gonzalez-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M. German. Macro-level software evolution: A case study of a large software compilation. Empirical Software Engineering, 14:262–285, 2009.

[133] Raymond Nguyen. Evolution and Architecture of Open Source Software Collections: A Case Study of Debian. University of Waterloo, August 2012.

[134] Théo Zimmermann and Jean-Rémy Falleri. A grounded theory of Community Package Maintenance Organizations. In Empirical Software Engineering, volume 28, page 101, 2023. doi: 10.1007/s10664-023-10337-4.

[135] Juan Mateos-Garcia and W. Edward Steinmueller. The institutions of open source software: Examining the Debian community. In Information Economics and Policy, volume 20 of Empirical Issues in Open Source Software, pages 333–344, December 2008. doi: 10.1016/j.infoecopol.2008.06.001.

[136] Bert M. Sadowski, Gaby Sadowski-Rasters, and Geert Duysters. Transition of governance in a mature open software source community: Evidence from the Debian case. In Information Economics and Policy, volume 20 of Empirical Issues in Open Source Software, pages 323–332, December 2008. doi: 10.1016/j.infoecopol.2008.05.001.

[137] Gregorio Robles, Jesus Gonzalez-Barahona, Grupo Sistemas, Rey Juan, Carlos Madrid, and Martin Michlmayr. Evolution of volunteer participation in libre software projects: Evidence from Debian. 2005.

[138] Christophe Lazaro. La Liberté Logicielle : Une Ethnographie Des Pratiques d'échange et de Coopération Au Sein de La Communauté Debian. Academia, 2008.

[139] Sara Schoonmaker. Hacking The Global: Constructing markets and commons through free software. In Information, Communication & Society, volume 15, pages 502–518, May 2012. doi: 10.1080/1369118X.2012.665938.

[140] E. Gabriella Coleman. Coding Freedom: The Ethics and Aesthetics of Hacking. Princeton University Press, 2013. ISBN 978-0-691-14460-3.

[141] Rebecca E. Grinter. Recomposition: Coordinating a Web of Software Dependencies. In Computer Supported Cooperative Work (CSCW), volume 12, pages 297–327, September 2003. doi: 10.1023/A:1025012916465.

[142] Daniel M. German, Jesus M. Gonzalez-Barahona, and Gregorio Robles. A Model to Understand the Building and Running Inter-Dependencies of Software. In 14th Working Conference on Reverse Engineering (WCRE 2007), pages 140–149, October 2007. doi: 10.1109/WCRE.2007.5.

[143] Maelick Claes, Tom Mens, Roberto Di Cosmo, and Jérôme Vouillon. A Historical Analysis of Debian Package Incompatibilities. In 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, pages 212–223, May 2015. doi: 10.1109/MSR.2015.27.

[144] Roberto Di Cosmo, Stefano Zacchiroli, and Paulo Trezentos. Package upgrades in FOSS distributions: Details and challenges. In Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades, HotSWUp '08, pages 1–5, New York, NY, USA, October 2008. Association for Computing Machinery. ISBN 978-1-60558-304-4. doi: 10.1145/1490283.1490292.

[145] Mike Hinchey. Intercomponent Dependency Issues in Software Ecosystems. In Software Technology: 10 Years of Innovation in IEEE Computer, pages 35–57. IEEE, 2018. ISBN 978-1-119-17422-6. doi: 10.1002/9781119174240.ch3.

[146] Ralf Treinen and Stefano Zacchiroli. Solving package dependencies: From EDOS to Mancoosi. arXiv, November 2008. doi: 10.48550/arXiv.0811.3620.

[147] Martin Michlmayr, Francis Hunt, and David Probert. Release Management in Free Software Projects: Practices and Problems. In Joseph Feller, Brian Fitzgerald, Walt Scacchi, and Alberto Sillitti, editors, Open Source Development, Adoption and Innovation, pages 295–300, Boston, MA, 2007. Springer US. ISBN 978-0-387-72486-7. doi: 10.1007/978-0-387-72486-7_31.

[148] E. Gabriella Coleman. Three Ethical Moments in Debian. Rochester, NY, September 2005. doi: 10.2139/ssrn.805287.

[149] E.G. Coleman and Benjamin Hill. The Social Production of Ethics in Debian and Free Software Communities: Anthropological Lessons for Vocational Ethics. In Stefan Koch, editor, Free/Open Source Software Development, pages 273–295. IGI Global, 2005. ISBN 978-1-59140-369-2 978-1-59140-371-5. doi: 10.4018/978-1-59140-369-2.ch013.

[150] David G. Messerschmitt and Clemens Szyperski. Software Ecosystem: Understanding an Indispensable Technology and Industry. The MIT Press, August 2003. ISBN 978-0-262-25666-7. doi: 10.7551/mitpress/6323.001.0001.

[151] Tom Mens, Maálick Claes, Philippe Grosjean, and Alexander Serebrenik. Studying Evolving Software Ecosystems based on Ecological Models. In Tom Mens, Alexander Serebrenik, and Anthony Cleve, editors, Evolving Software Systems, pages 297–326. Springer, Berlin, Heidelberg, 2014. ISBN 978-3-642-45398-4. doi: 10.1007/978-3-642-45398-4_10.

[152] Mircea Lungu. Towards reverse engineering software ecosystems. In 2008 IEEE International Conference on Software Maintenance, pages 428–431, September 2008. doi: 10.1109/ICSM.2008.4658096.

[153] Kathleen Schenkel, Selena Bliesener, Angela Calabrese Barton, and Edna Tan. Community Ethnography. In Science Scope, volume 43, pages 56–67. National Science Teachers Association, 2020.

[154] Virginia Braun and Victoria Clarke. Thematic Analysis: A Practical Guide. SAGE, Los Angeles London New Delhi Singapore Washington DC Melbourne, 2022. ISBN 978-1-4739-5323-9 978-1-4739-5324-6.

[155] Emeline Brulé. Thematic analysis in HCI. https://medium.com/usabilitygeek/thematic-analysis-in-hci-57edae583ca9, January 2021.

[156] Raymond Nguyen and Ric Holt. Life and death of software packages: An evolutionary study of Debian. In Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '12, pages 192–204, USA, November 2012. IBM Corp.

[157] H. Karasti and K.S. Baker. Infrastructuring for the long-term: Ecological information management. In 37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of The, pages 10 pp.–, 2004. doi: 10.1109/HICSS.2004.1265077.

[158] Chris Lamb and Stefano Zacchiroli. Reproducible Builds: Increasing the Integrity of Software Supply Chains. In IEEE Software, volume 39, pages 62–70, March 2022. doi: 10.1109/MS.2021.3073045.

[159] Roberto Di Cosmo and Stefano Zacchiroli. Software Heritage: Why and How to Preserve Software Source Code. In iPRES 2017 Conference Proceedings. 2017.

[160] Roberto Di Cosmo. Archiving and Referencing Source Code with Software Heritage. In Anna Maria Bigatti, Jacques Carette, James H. Davenport, Michael Joswig, and Timo Wolff, editors, Mathematical Software – ICMS 2020, pages 362–373, Cham, 2020. Springer International Publishing. ISBN 978-3-030-52200-1. doi: 10.1007/978-3-030-52200-1_36.

[161] Roberto Di Cosmo and Stefano Zacchiroli. The Software Heritage Open Science Ecosystem. In Tom Mens, Coen De Roover, and Anthony Cleve, editors, Software Ecosystems: Tooling and Analytics, pages 33–61. Springer International Publishing, Cham, 2023. ISBN 978-3-031-36060-2. doi: 10.1007/978-3-031-36060-2_2.

[162] Jonne Niemelä. A Systematic Mapping Study of Crunch Time in Video Game Development. Pro gradu -työ, J. Niemelä, June 2021.

[163] Daeana Paula Bourscheid, Andrea Valéria Steil, and Elizabet da Silva Paixão. Crunch on video game production: Practices to avoid for healthier workplaces in Game Development. In Proceedings of the 20th International Conference on the Foundations of Digital Games, FDG '25, pages 1–5, New York, NY, USA, May 2025. Association for Computing Machinery. ISBN 979-8-4007-1856-4. doi: 10.1145/3723498.3723730.

[164] Lee Vinsel and Andrew L. Russell. The Innovation Delusion: How Our Obsession with the New Has Disrupted the Work That Matters Most. Currency, New York, first edition edition, 2020. ISBN 978-0-525-57569-6.

[165] Paul E. Ceruzzi. A History of a Modern Computing. Mit Press, 1998. ISBN 978-0-262-03255-1.

[166] Sacha Loeve. I. La Loi de Moore : enquête critique sur l'économie d'une promesse. In Sciences et technologies émergentes : pourquoi tant de promesses ?, pages 91–113. Hermann, 2015. ISBN 978-2-7056-9106-6. doi: 10.3917/herm.josep.2015.01.0091.

[167] Sacha Loeve. La Loi de Moore, entre anticipation technologique et économie de la promesse | Cahiers Costech. https://www.costech.utc.fr/CahiersCostech, (2020-02-13 15:55:22|affdate{Y-m-d}). doi: 10.34746/cahierscostech85.

[168] G.E. Moore. Cramming More Components Onto Integrated Circuits. In Proceedings of the IEEE, volume 86, pages 82–85, January 1998. doi: 10.1109/JPROC.1998.658762.

[169] Gordon E. Moore. Progress in digital integrated electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]. In IEEE Solid-State Circuits Society Newsletter, volume 11, pages 36–37, September 2006. doi: 10.1109/N-SSC.2006.4804410.

[170] Gordon E. Moore. The microprocessor: Engine of the technology revolution. In Commun. ACM, volume 40, pages 112–114, February 1997. doi: 10.1145/253671.253746.

[171] Jerry Wu, Yin-Lin Shen, Kitt Reinhardt, Harold Szu, and Boqun Dong. A Nanotechnology Enhancement to Moore⁄s Law. In Applied Computational Intelligence and Soft Computing, volume 2013, page 426962, 2013. doi: 10.1155/2013/426962.

[172] Ray Kurzweil. The Singularity Is Near: When Humans Transcend Biology. Viking, 2005. ISBN 978-0-670-03384-3.

[173] Naomi Klein and Astra Taylor. End Times Fascism and the Fight for the Living World. Farrar, Straus and Giroux. ISBN 978-0-374-62140-7.

[174] Jérôme Denis and David Pontille. Le soin des choses. Politiques de la maintenance. La Découverte, October 2022.

[175] James W. Cortada. Change and Continuity at IBM: Key Themes in Histories of IBM. In The Business History Review, volume 92, pages 117–148. Cambridge University Press, 2018.

[176] Watts S. Humphrey. Software Unbundling: A Personal Perspective. In IEEE Ann. Hist. Comput., volume 24, pages 59–63, January 2002. doi: 10.1109/85.988582.

[177] Katharina Pistor. The Code of Capital: How the Law Creates Wealth and Inequality. Princeton University Press, May 2019. ISBN 978-0-691-18943-7.

[178] James Boyle. The Public Domain: Enclosing the Commons of the Mind. Yale University Press, January 2008. ISBN 978-0-300-13740-8.

[179] Philippe Aigrain. Cause commune: l'information entre bien commun et propriété. publie.net, January 2013. ISBN 978-2-8145-0659-6.

[180] Lionel Maurel. Comprendre les risques d'enclosure des communs de la connaissance. In La vie des idées, September 2015.

[181] Cory Doctorow. Enshittification: Why Everything Suddenly Got Worse and What To Do About It. Verso Books, October 2025. ISBN 978-1-83674-223-4.

[182] Breaking Free: Pathways to a fair technological future. Technical report, FORBRUKERRÅDET The Norwegian Consumer Council, 2026.

[183] Guillaume Pitron. The Dark Cloud: How the Digital World Is Costing the Earth. Scribe Publications, May 2023. ISBN 978-1-76138-510-0.

[184] Jean-Baptiste Fressoz. More and More and More: An All-Consuming History of Energy. HarperCollins, August 2025. ISBN 978-0-06-344496-6.

[185] Anaëlle Beignon, Thomas Thibault, and Nolwenn Maudet. Imposing AI: Deceptive design patterns against sustainability. In Proceedings of 11th Workshop on Computing Within Limits, June 2025. doi: 10.48550/arXiv.2508.08672.

[186] Florence Maraninchi. Pourquoi je n'utilise pas ChatGPT. Academia, February 2025. ISSN 2265-2434. doi: 10.58079/1382x.

[187] Sebastien Shulz, Mathieu O'neil, Sébastien Broca, and Angela Daly. Digital Commons for the Ecological Transition: Ethics, Praxis and Policies. TripleC: Communication, Capitalism & Critique, 22(1):348–365, April 2024. doi: 10.31269/triplec.v22i1.1456.

Appendix A

Selected conference talks related to Android

Supplementary table 1: selected conference talks related to the study of Android					
#	**Conference**	**Talk**	**Author**	**Function**	**Date**
1	Seminar Politiques environnementales du numérique	Fairphone OS maintenance	Agnès Crépet	Head of Software Longevity & Information Technology at Fairphone	2022
2	South Tyrol Free Software Conference 2019	MicroG - what it is and where it's going	Marvin Wissfeld	main developer of microG and LineageOS for microG	2019
3	Capitole du libre 2023	Open Source for Sustainable and Long lasting Phones	Agnès Crépet & Luca Weiss	Fairphone maintenance team	2023
4	Capitole du libre 2023	Mainline Linux on Fairphone? Yes, please!	Luca Weiss	Android engineer at Fairphone, PostmarketOS and Linux kernel developer	2023
5	Android MicroConference @ Linux Plumbers Conference	Entire sessions of 2018, 2024	Various	Developers from Google Android and Linux kernel	2018 to 2024
6	Kernel Recipes	Selected conferences on mainlining, upstreaming and Android	Various	2017, 2022 and 2024 editions	2017 to 2024
7	Open source firmware conference	Selected conferences on open source and industrial firmware maintenance	Various	2023 and 2024 editions	2023, 2024
8	miniDebConf Cambridge	Trixie on mobile: are we there yet?	Arnaud Ferraris	Mobian team leader	2024
9	miniDebConf Toulouse	Using Debian on mobile phones (BoF)	Arnaud Ferraris	Mobian team leader	2024
10	Debian conference 2023	The year of Linux on Desktop Mobile	Evangelos Ribeiro Tzaras	Mobian and PureOS developer	2023
11	Debian conference 2023	How many mobile linux developers does it take?	Evangelos Ribeiro Tzaras	Mobian and PureOs developer	2023
12	Debian conference 2023	Sustainable computing using old smartphones	Suraj Kumar Mahto	FOSS contributor and evangelist (Gnome, KDE)	2023
13	Free Silicon Conference	Sustainability sessions in 2023 and 2024	Various	Developers of open source tools to design & manufacture chips	2024