Compas'2023 : Parallélisme/ Architecture/ Système
LISTIC/USMB - Annecy France, 4-7 juillet 2023

Comment rater la reproductibilité de ses expériences ?

Quentin Guilloteau[1], Adrien Faure[1], Millian Poquet[2], Olivier Richard[1]

[1]Univ. Grenoble Alpes, INRIA, CNRS, LIG, Grenoble F-38000
Prénom.Nom@univ-grenoble-alpes.fr
[2]IRIT, Université de Toulouse, France
Millian.Poquet@irit.fr

Résumé
La communauté informatique commence doucement à se préoccuper des problèmes de reproductibilité de ses expériences. Le nombre de contributions sous le label "reproductibilité" est en croissance et des conférences demandent maintenant une évaluation des artéfacts par les pairs. Cependant, nous pensons que les problématiques qui y sont adressées ne sont que partiellement comprises ou utiles à la cause. La contribution principale de ce papier est l'exposition des imperfections en termes de reproductibilité des pratiques actuelles et des outils usuels pour la mise en place d'un environnement logiciel pour une expérience. Nous présentons ensuite une méthodologie se basant sur des gestionnaires de paquets fonctionnels pour favoriser une bonne reproductibilité de l'environnement logiciel.

Mots-clés : Reproductibilité, Environnement logiciel, Gestionnaires de paquets fonctionnels

---

1. Introduction

La communauté scientifique traverse une crise de la reproductibilité depuis une dizaine d'années et l'informatique n'y fait pas exception [25, 2]. La reproductibilité des travaux est essentielle pour construire des connaissances robustes, elle permet d'augmenter la fiabilité des résultats en limitant le nombre de biais méthodologiques et d'analyses ainsi que de favoriser l'échange de connaissances fiables. En 2015, Collberg et al. [9] ont étudié la reproductibilité de 402 articles expérimentaux publiés à des conférences et journaux en *système*. Chaque article étudié comportait un lien vers le code source qui a permis de réaliser l'expérience de l'article. Sur ces 402 papiers, 46% n'étaient pas reproductibles. Les causes étaient principalement que *(i)* le code n'était pas accessible *(ii)* le code ne compilait pas ou ne s'exécutait pas *(iii)* les expériences nécessitaient du matériel spécifique.
Pour mettre en avant les travaux reproductibles plusieurs éditeurs (comme ACM ou Springer) ont mis en place un système d'évaluation des artéfacts liés à une publication. Si, lors de l'évaluation par les pairs un papier a pu être reproduit, la publication aura un ou plusieurs badges correspondants au niveau de reproductibilité atteint.
La notion de reproductibilité est souvent utilisée au sens large et regroupe plusieurs concepts. La terminologie utilisée dans la suite de ce document est celle définie par l'ACM pour la validation d'artéfact [1], et comporte trois niveaux de reproductibilités : *(1)* la *répétabilité* : les mesures peuvent être ré-obtenues par les personnes originaires du travail, *(2)* la *reproductibilité* : les mesures peuvent être ré-obtenues par des personnes étrangères au travail initial avec les artéfacts

mis à disposition par les auteurs *(3)* la *réplicabilité* : les mesures peuvent être ré-obtenues par des personnes étrangères au travail et ne bénéficiant d'aucun artéfact du travail de base.
L'évaluation d'artéfacts est un point important qui permet de garantir la reproduction des expériences, mais elle n'est pas suffisante. Bien qu'être capable de reproduire une expérience soit un gage de validation scientifique, l'expérience et son environnement sont souvent trop rigides pour être approfondis par des tiers, voire par les auteurs eux-mêmes. Nous pensons que la **reproduction avec variation** devrait être appuyée par la communauté. Variation signifie ici qu'un tiers est capable de modifier l'environnement des auteurs pour approfondir des travaux ou investiguer d'autres pistes de recherche. Pour cela, les environnements matériels, logiciels et scripts d'expériences doivent être correctement définis et facilement modifiables.
Ce papier se concentre sur l'environnement logiciel. Pour une vision d'ensemble des problèmes de reproductibilité, nous redirigeons les lecteurs vers [18]. La section 2 présente la motivation de ce papier. Nous exposons en section 3 plusieurs pièges fréquemment rencontrés pouvant faire échouer la reproductibilité de l'environnement logiciel d'expériences. Enfin en section 4, nous motivons l'usage de gestionnaires de paquets fonctionnels comme une solution à la majorité des problèmes de reproductibilité.

2. Contexte et Motivation

Imaginons que dans votre enfance votre grand-mère ait cuisiné un délicieux gâteau au chocolat et que vous voulez apprendre à le faire vous-même, car vous pensez pouvoir l'améliorer et le rendre encore meilleur. Vous pouvez essayer de reproduire le gâteau avec votre lointain souvenir d'enfance et votre intuition culinaire, mais le résultat risque d'être décevant. Vous pouvez également demander la recette à vos parents s'ils s'en rappellent, mais il se peut que vous ne receviez que des instructions vagues. Peut-être que votre grand-mère avait juste improvisé la recette ! En fouillant dans un vieux livre de cuisine vous avez trouvé un morceau de papier avec ce qui ressemble à une recette de gâteau écrite de la main de votre grand-mère ! La recette est claire, bien détaillée et contient les quantités pour chaque ingrédient, l'ordre des différentes étapes, les temps que cuisson, etc. Vous suivez tout à la lettre, mais le résultat n'est pas identique… Peut-être n'avez-vous pas choisi le bon type d'œufs, ou que le modèle de votre four est diffèrent de celui de votre grand-mère, comment savoir ?
Une expérience sans l'environnement dans lequel elle a été exécutée la rend plus difficile à reproduire : des effets de bords liés à l'environnement peuvent apparaître. Il est facile d'oublier d'inclure dans l'environnement un élément qui impacte les performances de l'expérience. Les performances, mais aussi les résultats d'une simple application C peuvent dépendre des options de compilations [28] ou encore de la quantité de variables d'environnement UNIX [22].
La plupart des solutions actuelles en termes de "reproductibilité" relèvent du stockage d'artéfacts (images systèmes, conteneurs, machines virtuelles) et du rejeu d'expériences [26, 5, 6]. Bien que ce soit une partie importante du spectre de la reproductibilité, rien ne garantit que l'environnement logiciel soit reconstructible dans le futur, et donc l'expérience re-jouable, si les artéfacts disparaissent.
L'étape d'évaluation des artéfacts pour les conférences, est faite peu de temps après leur construction initiale, il est donc probable qu'elle soit exécutée avec des miroirs de paquets (`apt`, `rpm`, etc.) dans un état similaire. Cependant, qu'adviendra-t-il de la reconstruction de l'environnement dans 1 an ? 5 ans ? 10 ans ? L'objectif de la recherche scientifique est de se baser sur des travaux solides pour continuer de progresser. Cette vision de "*reproductibilité à court terme*" est un frein majeur au progrès scientifique et en opposition à sa philosophie.
Nous pensons que la notion qu'il faudrait mettre en avant est celle de la **variation** [21, 12]. C'est-

à-dire la possibilité pour un tiers d'utiliser l'environnement défini pour une expérience, dans le but d'investiguer une autre piste de recherche. Un exemple de variation pourrait être de changer l'implémentation de MPI utilisée (*e.g.,* MPICH au lieu de OpenMPI). Introduire une telle variation n'est possible que si l'environnement de base est correctement défini et modifiable.

3. Pièges usuels de la reproductibilité de l'environnement logiciel

3.1. Partage de l'environnement

Une manière évidente de rater la reproductibilité de ses expériences est de ne pas partager les environnements utilisés, ou de les partager à des endroits non pérennes. Des plateformes telles que Zenodo [30] ou Software-Heritage [17] permettent de stocker les artéfacts (scripts, environnements, etc.) de manière permanente.

3.2. Connaissance de l'environnement

Dans le cas où l'environnement logiciel ne contient que des paquets Python, geler les dépendances avec `pip` (`pip freeze`) n'est pas suffisant. `pip` ne décrit que l'environnement Python et ignore les dépendances *systèmes* qu'ont de nombreux paquets Python. Bien que recréer un environnement Python depuis un `requirements.txt` soit simple, installer une liste de paquets systèmes avec des versions spécifiques est en revanche bien plus complexe. De plus, lister les paquets systèmes à la main est sujet à erreurs et oublier un paquet est facile.
Des outils tels que Spack [13] ont une approche similaire à `pip`, mais également pour les paquets systèmes ainsi que leurs dépendances. Il est possible d'exporter l'environnement en format texte et de le reconstruire sur une autre machine. Cependant, l'environnement risque de ne pas être identique. En effet, Spack utilise des applications déjà présentes sur la machine pour reconstruire les paquets depuis les sources, et notamment un compilateur C. Ainsi, si différentes machines ont des compilateurs C différents, alors l'environnement reconstruit différera de l'environnement désiré. Un avantage de Spack est l'introduction et la gestion de variation.
Ces solutions se préoccupent uniquement de la pile logicielle au-dessus du système d'exploitation. Cependant, les résultats des expériences pourraient également dépendre du noyau, des pilotes… Ainsi, il est aussi important de bien capturer *l'entièreté* de la pile logicielle.

Piège 1: Capturer partiellement l'environnement logiciel de l'expérience.

Vos expériences peuvent ne pas seulement dépendre de paquets Python, mais aussi de paquets systèmes, voire du noyau. Il faut capturer l'environnement *en entier*.

Usuellement, la capture de toute la pile logicielle passe par son encapsulation dans une image système qui sera ensuite déployée pour exécuter les expériences. Une manière de générer une image système est de partir d'une image de base, de la déployer, d'exécuter les commandes nécessaires pour la mise en place de l'environnement et ensuite de la compresser. Des plateformes comme Grid'5000 [3] et Chameleon [20] proposent à leurs utilisateurs ce genre d'outils (`tgz-g5k` [14] et `cc-snapshot` [8] respectivement). Dans le cadre de la répétabilité et réplicabilité, si l'image reste disponible alors cette manière de produire une image est passable. En revanche, en ce qui concerne la traçabilité, vérifier les commandes qui ont été exécutées pour créer l'environnement repose uniquement sur la documentation de l'expérimentateur. De plus, une image n'est pas adaptée au versionnage, car dans un format binaire. Dans le cas où l'image n'est plus disponible, reconstruire l'image exacte est complexe et l'introduction précise de variation est utopique.

3.3. Dépendance à un état extérieur

Une manière de déployer un environnement complet pourrait être de le reconstruire à chaque expérience. EnOSlib [7] est une bibliothèque Python pour gérer des expériences distribuées et elle intègre un mécanisme d'installation de paquets de manière programmatique. Les utilisateurs peuvent ainsi en exécutant leurs scripts se baser sur un environnement par défaut qu'ils vont ensuite modifier pour le besoin de leurs expériences. Cette stratégie de partir d'un environnement par défaut a l'avantage qu'il est plus difficile d'oublier de spécifier une dépendance système dans la description de l'environnement, car son absence serait remarquée à chaque exécution. Cependant, il est toujours possible de se tromper de plusieurs façons. La reproductibilité des expériences utilisant ces solutions dépend fortement des environnements de base sur lesquels elles sont exécutées. Ces environnements de base sont gérés par les administrateurs de la plateforme et ont également une durée de vie, ce qui pose la question de leur pérennité. Supposons que ces administrateurs mettent à jour la version de l'environnement de base, alors qu'advient-il de la reproductibilité de l'environnement ? Pour ne pas dépendre de ces environnements, il est possible de fournir une image système à EnOSlib que la bibliothèque déploiera avec Kadeploy [19]. Cette image système doit alors pouvoir être accessible de manière pérenne et reconstructible. De plus, il est aussi primordial que des solutions comme EnOSlib soient correctement capturées dans l'environnement logiciel.

Piège 2: Oublier de capturer l'environnement logiciel du moteur d'expérience.

Capturer l'environnement logiciel dans lequel le moteur d'expérience est exécuté est tout aussi important que l'environnement de l'expérience elle-même.

Une meilleure approche pour la génération d'images est de les créer via des *recettes*. Ces recettes, comme les `Dockerfile` pour les conteneurs Docker ou les recettes Kamelon [27] pour des images systèmes, sont une suite de commandes à exécuter sur une image de base pour générer l'environnement désiré. Le format texte de ces recettes les rend facilement versionnables, partageables et (presque) reconstructibles. Ces images de base ont plusieurs versions qui sont identifiées par des labels (*tags*). Dans le cas de Docker, le tag de la dernière version est souvent appelée `latest`. Se baser sur cette version de l'image de base casse la traçabilité, et donc la reconstruction de l'image, car si lors d'une reconstruction future, il y a une nouvelle version plus récente de l'image de base, alors l'environnement ne se basera pas sur l'image initialement désirée. Une question importante est aussi de savoir si l'image de base est elle-même reconstructible et si ce n'est pas le cas, quelle est la pérennité des plateformes les hébergeant ?

Piège 3: Se baser sur des images non reproductibles.

Attention à la pérennité des images sur lesquelles vous vous basez. Par transitivité, si ces images de base ne sont pas reproductibles, alors la vôtre ne l'est pas non plus.

Un autre problème usuel est que la recette effectue une mise à jour du miroir (*e.g.*, `apt get update`) avant d'installer les paquets nécessaires pour l'environnement. Ceci a la fâcheuse propriété de ne pas rendre l'image reconstructible, car l'environnement dépend alors de l'état d'une entité extérieure qui est difficilement contrôlable. Afin d'être certain d'utiliser exactement les mêmes paquets à chaque reconstruction, une solution serait d'utiliser une capture (*snapshot*) du miroir [1]. EnOSlib permet de prendre en compte cette capture et de l'utiliser dans la suite du script d'expérience. En ce qui concerne l'introduction de variation, se baser sur un *snapshot* est

1. Exemple pour `debian` : `http://snapshot.debian.org/`

contraignant et peut rendre impossible d'installer certaines versions de paquets, ou peut créer des conflits avec d'autres paquets déjà installés.
Lorsque la recette doit télécharger un objet depuis l'extérieur, il est important de vérifier que l'objet obtenu est bien celui attendu. Dans le cas contraire, l'image a une dépendance à l'état de la source au moment de la construction. Ainsi, si la recette effectue un appel à `curl` pour récupérer un fichier de configuration ou un *snapshot* de miroir par exemple, la recette doit également s'assurer que le contenu du fichier est bien celui attendu. La pratique usuelle est de comparer le hash cryptographique de l'objet téléchargé avec celui de l'objet attendu.
Une problématique similaire survient lorsque la recette doit télécharger un paquet via `git` et le construire depuis les sources. Dans ce cas, il est primordial de bien fixer le commit utilisé dans la recette de l'image. En effet, ne pas connaître le commit utilisé revient à avoir une dépendance sur l'état du dépôt du paquet au moment de la construction de l'image (dernier commit sur la branche principale). Fixer le commit permet de connaître exactement les sources utilisées, ainsi que la possibilité d'introduire facilement de la variation avec un autre commit si besoin.

Piège 4: Ne pas vérifier le contenu des objets téléchargés.

Chaque objet provenant de l'extérieur doit être examiné pour être certain que son contenu soit celui attendu. Il est plus important que l'image ne soit plus constructible si l'objet téléchargé est différent, plutôt que de construire l'image avec une version de l'objet non attendue.

4. Méthodologie et outils pour éviter ces pièges

4.1. Gestionnaires de paquets fonctionnels

Des outils comme Nix [11] ou Guix [10] règlent la plupart des problèmes et erreurs énoncés dans la section précédente. Dans la suite de cette section, nous nous concentrerons sur Nix.
Nix est un gestionnaire de paquets fonctionnel pur visant la reproductibilité des paquets. Un paquet est défini comme une fonction dont les entrées sont les dépendances du paquet et les options de construction du paquet et le corps de la fonction contient les commandes pour le construire. La construction des paquets s'effectue ensuite dans une *sandbox* qui garantit une construction dans un environnement strictement contrôlé. Dans un premier temps, les sources sont téléchargées et leur contenu est vérifié par Nix. Si le hash des sources n'est pas celui attendu (`sha256` dans le listing 1), Nix stoppe la construction du paquet. Nix fait de même pour les dépendances du paquet **et récursivement**. Les commandes de construction du paquet sont exécutées dans l'environnement défini par l'utilisateur (`buildInputs` dans le listing 1), et aucun accès réseau ou au filesystem n'est possible.
Nix peut générer des environnements d'exécution, qui correspondent à des `virtualenv`s multi-langages, des images conteneurs (Docker, Singularity, LXC, etc.), des machines virtuelles ou des images systèmes. Le processus de construction d'un environnement est souvent itératif et laborieux. Construire des images avec Nix se fait de manière *déclarative* et a l'avantage d'être rapide lors d'une reconstruction avec modification [15], ainsi que de ne pas nécessiter d'optimisation dans l'ordre des opérations dans la recette. Comme les paquets de Nix sont des fonctions, introduire une variation revient simplement à modifier un argument lors de l'appel de la fonction. Dans le listing 1, il est facilement possible d'utiliser une autre version de `simgrid` lors de l'appel à la fonction construisant le paquet. Des systèmes comme `debian` stockent tous les paquets installés dans les répertoires `/usr/bin` et `/usr/lib` ce qui peut amener à des conflits de versions, limitant ainsi l'introduction de variation dans l'environne-

ment. En revanche, Nix crée un répertoire par variante de paquet, préfixé par le hash de ses sources qui est stocké dans un répertoire parent accessible en lecture seule appelé le *Nix Store* (`/nix/store`). Ainsi si une version différente d'un paquet est installée, alors le hash de ses sources sera diffèrent du hash de la version déjà installée et sera donc stockée dans un répertoire différent. L'avantage de cette méthode est l'isolation fine des paquets et la définition *précise* de la variable d'environnement `$PATH` pour générer des environnements logiciels. Cette méthode évite également la duplication des paquets dans le *store* : plusieurs programmes exécutables utilisant la même version de la même bibliothèque dynamique *renverront* vers le même fichier `.so`. La définition des paquets par fonctions facilite leur partage et distribution. Il existe une base de définitions de paquets communautaire de référence appelée `nixpkgs` [23], mais il est également possible pour des équipes ou groupes de recherche d'avoir leur base de paquets indépendamment. Guix-HPC [16] ou NUR-Kapack [24] en sont des exemples dans le contexte des systèmes distribués.

4.2. Limites des gestionnaires de paquets fonctionnels

Bien que des outils comme Nix et Guix aident grandement à la reproductibilité des environnements logiciels, il est toujours possible de rendre la recette impure et de dépendre d'un état de la machine hôte. Nix étudie ce problème avec la fonctionnalité expérimentale *Flake* [29].

Pour assurer la reproductibilité et traçabilité d'un environnement, Nix requiert que tous les paquets, ainsi que toutes les dépendances, aient leur code open source et qu'elles soient paquetées avec Nix. Ceci peut sembler contaminant et limitant dans le cas de logiciels propriétaires où le code source n'est pas disponible, tels qu'un compilateur Intel par exemple. Il est tout de même possible d'utiliser des applications propriétaires avec le mode `impure` de Nix, ce qui réduit la traçabilité et donc la reproductibilité de l'environnement logiciel.

La construction des paquets dans une *sandbox* passe par un mécanisme d'isolation du système de fichiers utilisant `chroot`. Historiquement, ceci n'était possible que si l'utilisateur a les droits `root`. Or, dans le cas de clusters de calculs ou plateformes expérimentales, ce genre de permissions limite l'adoption de Nix ou Guix. Cependant, la fonctionnalité *user namespace* du noyau Linux permet de passer outre ces besoins de droits dans une majorité des cas.

Comme Nix doit recompiler depuis les sources les paquets ne se trouvant pas dans le cache binaire, il se peut qu'une reconstruction future soit impossible, car l'hébergeur des sources n'existe plus [4]. Cependant, comme Software Heritage réalise des archives fréquentes de dépôts ouverts, il sera possible de retrouver les sources.

Ces outils demandent également un changement de point de vue sur la manière de gérer un environnement logiciel qui peut rendre la courbe d'apprentissage intimidante.

5. Conclusion

La communauté informatique commence à s'intéresser aux problématiques liées à la reproductibilité des expériences. Cependant, les problématiques sont peu comprises en profondeur. Faire des expériences reproductibles peut s'avérer *extrêmement* complexe. La gestion d'environnement est une illustration de cette complexité. Les outils usuels (`pip`, `spack`, `docker`, etc.) ne répondent pas à ces problématiques sans un effort conséquent des expérimentateurs et ne permettent qu'une "*reproductibilité à court terme*". Le *graal* de la reproductibilité est l'introduction précise de variation dans un environnement d'un tiers. Ce besoin de variation permet de se reposer sur des contributions solides afin de poursuivre la recherche scientifique proprement. Bien qu'il n'existe pas encore de solution parfaite, la nécessité d'un changement de pratique sur les problématiques de reproductibilité se doit d'être considérée.

A. Exemple de paquet Nix

```
1  { stdenv, fetchgit, simgrid, boost, cmake }:          # Dependances globales
2
3  stdenv.mkDerivation {                                 # Definition d'un paquet
4    name = "chord";                                     # Nom du paquet
5    version = "0.1.0";                                  # Version
6    src = fetchgit {                                    # Telechargement des sources
7      url = "https://gitlab.fr/me/chord";               # URL du depot
8      rev = "069d2a5bfa4c40...";                        # Commit a utiliser
9      sha256 = "sha256-ff4f...";                        # Hash des sources attendu
10   };
11   buildInputs = [ simgrid boost cmake ];              # Dependances pour le build
12 }
```

Listing 1 – Exemple de définition d'un paquet avec Nix. Les dépendances sont listées en première ligne. Le contenu du code est vérifié en ligne 9.

Références

[1] ACM. *Artefact review badging*. https://www.acm.org/publications/policies/artifact-review-badging. Accessed : 2023-04-04.

[2] Monya BAKER. « 1,500 scientists lift the lid on reproducibility ». In : *Nature* 533.7604 (mai 2016), p. 452-454. ISSN : 0028-0836, 1476-4687. DOI : 10.1038/533452a. URL : http://www.nature.com/doifinder/10.1038/533452a (visité le 03/05/2019).

[3] Daniel BALOUEK et al. « Adding Virtualization Capabilities to the Grid'5000 Testbed ». In : *Cloud Computing and Services Science*. Sous la dir. d'Ivan I. IVANOV et al. T. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, p. 3-20. ISBN : 978-3-319-04518-4. DOI : 10.1007/978-3-319-04519-1_1.

[4] BLINRY. *Building 15-year-old software with Nix*. https://blinry.org/nix-time-travel/. Accessed : 2023-04-16.

[5] Grant R BRAMMER et al. « Paper mâché : Creating dynamic reproducible science ». In : *Procedia Computer Science* 4 (2011), p. 658-667.

[6] Adam BRINCKMAN et al. « Computing environments for reproducibility : Capturing the "Whole Tale" ». In : *Future Generation Computer Systems* 94 (2019), p. 854-867.

[7] Ronan-Alexandre CHERRUEAU et al. « EnosLib : A Library for Experiment-Driven Research in Distributed Computing ». en. In : *IEEE Transactions on Parallel and Distributed Systems* 33.6 (juin 2022), p. 1464-1477. ISSN : 1045-9219, 1558-2183, 2161-9883. DOI : 10.1109/TPDS.2021.3111159. URL : https://ieeexplore.ieee.org/document/9534688/ (visité le 22/11/2021).

[8] Chameleon CLOUD. *The cc-snapshot utility*. https://chameleoncloud.readthedocs.io/en/latest/technical/images.html#the-cc-snapshot-utility. Accessed : 2023-04-03.

[9] Christian COLLBERG, Todd PROEBSTING et Alex M WARREN. « Repeatability and Benefaction in Computer Systems Research - A Study and a Modest Proposal ». en. In : (2015), p. 68.

[10] Ludovic COURTÈS. « Functional Package Management with Guix ». en. In : *arXiv :1305.4584 [cs]* (mai 2013). URL : http://arxiv.org/abs/1305.4584 (visité le 13/06/2020).

[11] Eelco DOLSTRA, Merijn de JONGE et Eelco VISSER. « Nix : A Safe and Policy-Free System for Software Deployment ». en. In : (2004), p. 14.

[12] Dror G. FEITELSON. « From Repeatability to Reproducibility and Corroboration ». en. In : *ACM SIGOPS Operating Systems Review* 49.1 (jan. 2015), p. 3-11. ISSN : 0163-5980. DOI : 10.1145/2723872.2723875. URL : https://dl.acm.org/doi/10.1145/2723872.2723875 (visité le 21/05/2020).

[13] Todd GAMBLIN et al. « The Spack package manager : bringing order to HPC software chaos ». en. In : *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Austin Texas : ACM, nov. 2015, p. 1-12. ISBN : 978-1-4503-3723-6. DOI : 10.1145/2807591.2807623. URL : https://dl.acm.org/doi/10.1145/2807591.2807623 (visité le 22/11/2021).

[14] GRID'5000. *Creating an environment images using tgz-g5k*. https://grid5000.fr/w/Environment_creation#Creating_an_environment_images_using_tgz-g5k. Accessed : 2023-04-03.

[15] Quentin GUILLOTEAU et al. « Painless Transposition of Reproducible Distributed Environments with NixOS Compose ». In : *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. 2022, p. 1-12. DOI : 10.1109/CLUSTER51413.2022.00051.

[16] GUIX-HPC. *Guix-HPC*. https://gitlab.inria.fr/guix-hpc/guix-hpc. Accessed : 2023-04-04.

[17] Software HERITAGE. *Software Heritage*. https://www.softwareheritage.org/. Accessed : 2023-03-30.

[18] Peter IVIE et Douglas THAIN. « Reproducibility in scientific computing ». In : *ACM Computing Surveys (CSUR)* 51.3 (2018), p. 1-36.

[19] Emmanuel JEANVOINE, Luc SARZYNIEC et Lucas NUSSBAUM. « Kadeploy3 : Efficient and scalable operating system provisioning for clusters ». In : *USENIX Association* 38.1 (2013), p. 38-44.

[20] Kate KEAHEY et al. « Lessons learned from the chameleon testbed ». In : *2020 USENIX annual technical conference (USENIX ATC 20)*. 2020, p. 219-233.

[21] Michael MERCIER, Adrien FAURE et Olivier RICHARD. « Considering the Development Workflow to Achieve Reproducibility with Variation ». In : *SC 2018-Workshop : ResCuE-HPC*. 2018, p. 1-5.

[22] Todd MYTKOWICZ et al. « Producing wrong data without doing anything obviously wrong ! » In : *ACM Sigplan Notices* 44.3 (2009), p. 265-276.

[23] NIXOS. *nixpkgs*. https://github.com/nixos/nixpkgs. Accessed : 2023-04-04.

[24] OAR-TEAM. *NUR-Kapack*. https://github.com/oar-team/nur-kapack. Accessed : 2023-04-04.

[25] David RANDALL et Christopher WELSER. *The Irreproducibility Crisis of Modern Science. Causes, Consequences, and the Road to Reform*. en. New York : National Association of Scholars, 2018. URL : https://www.nas.org/reports/the-irreproducibility-crisis-of-modern-science.

[26] Daniel ROSENDO et al. « E2Clab : Exploring the Computing Continuum through Repeatable, Replicable and Reproducible Edge-to-Cloud Experiments ». In : *Cluster 2020 - IEEE International Conference on Cluster Computing*. Kobe, Japan, sept. 2020, p. 1-11. DOI : 10.1109/CLUSTER49012.2020.00028. URL : https://hal.science/hal-02916032.

[27] Cristian RUIZ et al. « Reconstructable Software Appliances with Kameleon ». en. In : *ACM SIGOPS Operating Systems Review* 49.1 (jan. 2015), p. 80-89. ISSN : 0163-5980. DOI : 10.1145/2723872.2723883. URL : https://dl.acm.org/doi/10.1145/2723872.2723883 (visité le 12/06/2020).

[28] Victoria STODDEN et Matthew S KRAFCZYK. « Assessing reproducibility : An astrophysical example of computational uncertainty in the HPC context ». In : *Proceedings of the 1st Workshop on Reproducible, Customizable and Portable Workflows for HPC at SC*. T. 18. 2018.

[29] TWEAG.IO. *What problems do flakes solve ?* https://www.tweag.io/blog/2020-05-25-flakes/. Accessed : 2023-04-04.

[30] ZENODO. *Zenodo*. https://zenodo.org/. Accessed : 2023-03-30.