

## Cycle de vie du logiciel

Pour accéder à la ressource : <https://doi.org/10.13143/N9JG-TT39>

Date de publication : 11/05/2026

### Sommaire

1.	Nouvelle question scientifique .....	3
2.	Travaux préliminaires .....	4
2.1	L'état de l'art .....	4
2.1.1	Objectifs de l'état de l'art .....	4
2.1.2	Dimensions de l'état de l'art .....	5
2.2	La valorisation .....	6
2.2.1	La valorisation au sens économique du terme.....	6
2.2.2	La valorisation auprès du monde académique.....	6
2.2.3	La valorisation auprès du monde social .....	7
2.3	Les licences .....	8
3.	Planification .....	8
3.1	Design du logiciel .....	8
3.2	Plan de gestion de logiciel (PGL) .....	9
3.3	Forge logicielle .....	10
3.4	Choix d'une licence.....	11
3.4.1	Licences permissives (sans copyleft) .....	12
3.4.2	Licences copyleft .....	12

3.4.3 Double licence .....	13
4. Développement .....	14
4.1 Les choix à faire .....	14
4.1.1 Langage de programmation .....	14
4.1.2 Modules ou classes.....	14
4.1.3 Systèmes et utilisateurs ciblés .....	14
4.1.4 Interfaces .....	15
4.2 Comment les collaborateurs vont-ils s'organiser pour développer le code ? ..	15
4.3 Documentation .....	15
5. Tests.....	16
5.1 Tests unitaires .....	16
5.2 Tests d'intégration .....	17
6. Intégration continue .....	18
7. Maintenance.....	19
7.1 Maintenance corrective .....	20
7.2 Maintenance perfective .....	20
8. Ouverture du code .....	20
9. Contributions.....	22
10. Rapport de bug.....	23
10.1 Recueillir des rapports de bugs complets.....	23
10.2 Trier et assigner les bugs.....	23
11. Publication et Archivage .....	24
11.1 Publier depuis la forge .....	24
11.2 Publier un software paper .....	25
11.2.1 Avantages de la publication d'un software paper .....	25
11.2.2 Où publier un software paper ?.....	26
11.3 Archiver .....	26
11.3.1 Présentation de Software Heritage .....	27

11.3.2	Comment archiver le code dans Software Heritage depuis la forge ?	27
11.4	Référencer.....	28

Cette ressource pédagogique est le fruit de la collaboration entre [Damien Belvèze](#), [Julien Caugant](#), [Maria Grazia Santangelo](#), [Jérôme Marini](#), [Monica Michel Rodriguez](#), [Marco Milanesio](#) et DoRANum.

Elle permet de répondre aux différentes questions soulevées à chaque étape du cycle de vie du logiciel.

## 1. Nouvelle question scientifique

À quelle problématique scientifique répond le logiciel ?

Le développement d'un logiciel de recherche ne commence pas directement par l'écriture de code. Il est précédé de plusieurs travaux essentiels qui permettent de clarifier les objectifs, de cadrer le projet et de garantir que l'outil développé sera pertinent, réutilisable et conforme aux besoins scientifiques. Tout commence par une **problématique scientifique claire**. Le logiciel est un outil au service de la recherche, il doit donc répondre à une question précise :

- quel phénomène cherche-t-on à observer, mesurer, modéliser ou analyser ?
- quels types de données ou d'interactions le logiciel devra-t-il traiter ?

Exemple : un chercheur en écologie souhaite développer un outil pour analyser les trajectoires migratoires des oiseaux à partir de données GPS.

Avant toute conception d'un nouveau logiciel, il est essentiel de faire **un état de l'art**, c'est-à-dire :

- recenser les outils existants (logiciels, bibliothèques, scripts, plateformes),
- évaluer leur pertinence, leurs limites, leurs licences et possibilités d'adaptation,
- identifier les lacunes que le futur logiciel pourrait combler.

## 2. Travaux préliminaires

Par où commencer ? Existe-t-il des travaux précédents ? Qu'utiliser et comment le faire ?

Dès lors que le protocole de l'expérimentation est défini et que l'on a une idée des fonctionnalités et objectifs principaux du code que l'on souhaite développer, il convient :

- **d'étudier dans la littérature scientifique** ce qui est fait dans le domaine et
- d'analyser comment le prendre en compte dans son projet.

À ces fins, il est nécessaire de faire d'abord un **état de l'art** pour déterminer :

- quels **méthodes et algorithmes** peuvent être mobilisés pour répondre au mieux à une question scientifique,
- quels **logiciels** implémentent la méthode ou l'algorithme choisi,
- s'il est possible d'adapter ces logiciels au contexte de son projet.

Il faut aussi réfléchir à la **licence** que ce logiciel utilise. En effet, cela pourrait impacter la manière dont le code peut être réutilisé.

### 2.1 L'état de l'art

#### 2.1.1 Objectifs de l'état de l'art

L'état de l'art a plusieurs objectifs clés :

- identifier les algorithmes et les méthodes susceptibles de nous aider à résoudre la question,
- identifier les outils logiciels qui implémentent ces méthodes et ces algorithmes en tout ou partie,
- évaluer les forces et les limites de ces outils : performances, compatibilité, facilité d'utilisation, licence, maintenabilité,
- repérer les manques ou les besoins non couverts qui justifieraient le développement d'un nouvel outil,

- s'inspirer des bonnes pratiques et des choix technologiques pertinents utilisés dans des projets similaires,
- justifier la pertinence du projet dans un contexte de recherche.

### 2.1.2 Dimensions de l'état de l'art

L'état de l'art peut couvrir plusieurs dimensions :

- **Littérature scientifique**

On procède, comme pour l'état de l'art des publications, en cherchant sur des bases de publications (Google Scholar, Matilda, Pubmed, Web of Science, Isidore ...) des articles sur des méthodologies, outils ou algorithmes existants. On inclut dans la littérature scientifique les *software journals* qui permettent de comprendre le fonctionnement et le potentiel d'un logiciel donné, comme le [Journal JOSS](#).

- **Outils et logiciels existants**

Il s'agit d'examiner ce qui a déjà été développé dans le domaine visé, de façon à ne pas réinventer la roue, mais aussi à mieux comprendre les références techniques, les solutions dominantes, et les limites actuelles du paysage logiciel.

On analyse quelles sont les capacités des outils et sur quels langages ou *frameworks* ils reposent (Python, R, C++, Java, etc.).

On vérifie la qualité du code et de la documentation. Le code est-il ouvert et bien structuré ?

Le code est-il open source ? Le logiciel peut-il s'intégrer facilement à d'autres outils ?

On peut chercher ces outils dans :

- des plateformes de codes open source : GitHub, GitLab, Bitbucket,
- des bibliothèques de packages : PyPI (Python), CRAN (R), npm (JavaScript), Bioconductor (bioinformatique), bibliothèques standards pour C selon les disciplines...

- **Bibliothèques de codes sources prêts au réemploi**

Lorsque l'on travaille dans des champs particuliers, il arrive que des bibliothèques de codes sources soient déjà prêtes à la consultation. On n'a plus qu'à repérer ce qu'il nous faut, le charger sous la forme de conteneur ou de paquet, l'installer et le modifier ou l'intégrer à son propre code source. Il faut vérifier que l'objet récupéré soit bien utilisable sur son propre environnement logiciel et matériel.

Exemples :

- Paquets mis à la disposition de la communauté scientifique qui travaille sous R : [RopenSci](#)
- Conteneurs en biologie : [BioContainers](#)
- Paquets mis à disposition des chercheurs sur [Guix-Science](#) (nécessitent l'usage du gestionnaire de paquets Guix)
- Applications utiles à la biologie et présentes dans le gestionnaire de package Conda : [Bioconda](#)
- Outils du Medialab Sciences Po pour le traitement des données en sciences sociales : [Medialab Sciences Po](#)
- L'Institut Pasteur donne par ailleurs [quelques pistes](#) pour localiser les codes sources et packages utiles à la recherche sur le web
- Le [catalogue des logiciels libres de la recherche académique](#) recense les logiciels libres issus de la recherche académique

## 2.2 La valorisation

La valorisation peut couvrir plusieurs sens :

### 2.2.1 La valorisation au sens économique du terme

Dans le cadre de la science ouverte, le but n'est pas de rendre un code source inaccessible pour en faire une exploitation commerciale. Même si une exploitation commerciale peut en être faite, d'une manière ou d'une autre, le **code** doit rester **FAIR** : trouvable, accessible, interopérable et réutilisable.

En savoir plus sur les [principes FAIR adaptés aux logiciels](#).

Toutefois, si une partie de la recherche est financée par un acteur privé (cas d'une thèse CIFRE par exemple) et si le logiciel est conçu pour répondre à un problème qui se pose à cet acteur industriel, il est probable que la propriété intellectuelle doive être partagée avec lui. C'est d'autant plus le cas si l'acteur privé contribue au code source qui sera créé. Ce contexte peut donc contraindre le choix de la licence par le chercheur.

### 2.2.2 La valorisation auprès du monde académique

Pour gagner de la valeur, le code doit être maintenu dans le temps ; tâche difficile et souvent impossible à faire pour un chercheur seul. Il faut donc convaincre la communauté des personnes qui utilisent son logiciel de contribuer à sa maintenance et à son développement.

Pour cela un **software paper** qui décrira le potentiel de réutilisation possible, est tout à fait indiqué. Lorsque les contributeurs deviennent plus nombreux, il faut mettre au point une gouvernance. Il faut aussi prendre des décisions collégiales afin que les contributions apportées au code initial ne s'écartent pas de ses fonctions premières et de son sens originel.

Pour en savoir plus sur les [software papers](#).

Penser à cette valorisation académique, intégrer une documentation riche, et citer les producteurs des codes réutilisés participe à la valorisation des pairs et à l'émergence des nouvelles idées scientifiques.

### 2.2.3 La valorisation auprès du monde social

Le souci de pérennité d'un logiciel n'implique pas seulement les autres chercheurs. Par exemple Zotero, est utilisé aujourd'hui en dehors de la recherche pour les rédacteurs ayant besoin de cette solution. Dans son ouverture au monde social, il a vu évoluer son modèle économique tout en restant entièrement libre et utilisable pour tous.

La valorisation d'un logiciel passe assez souvent par l'application d'une **licence duelle (dual license)** sur le produit :

- d'une part le logiciel reste accessible et gratuit pour tous les utilisateurs sous une licence restrictive (GPL),
- d'autre part les services de valorisation permettent à des entreprises de négocier l'achat de versions plus simples pour exploiter le logiciel sans avoir à délivrer le code propriétaire dans lequel celui-ci s'insère. Ce qui est normalement exigé dans le cadre d'une licence contaminante type GPL. Le type de base SQL MariaDB est un bon exemple d'application de ce type de licence duelle.

Un logiciel produit par un chercheur sur son temps de travail ne saurait être mis à disposition sous forme payante à l'ensemble des publics. En effet, le **chercheur a la paternité du code source** mais **sa propriété revient à son employeur**. Celui-ci doit s'assurer que le code source produit sera au moins mis à disposition gratuitement pour les individus.

#### Pour aller plus loin

Blanc, I., Boulet, P., Courbebaisse, G., Daydé, M., Gérard, S., et al. Production et valorisation des logiciels issus de la recherche publique française. Comité pour la science ouverte. 2024. [hal-04844037](#)

## 2.3 Les licences

Si l'on souhaite réutiliser un logiciel lors d'un développement, il est essentiel de consulter la licence attribuée au logiciel.

Les licences sont des **contrats de mise à disposition du logiciel**. Elles précisent les droits conférés en matière d'adaptation, de modification ou de redistribution du code source.

En l'absence de licence clairement définie, le logiciel est juridiquement considéré comme "tous droits réservés", ce qui empêche légalement toute réutilisation, même dans un cadre académique. S'il n'y a pas de droit explicitement donné à travers une licence, utiliser un logiciel relève de la contrefaçon.

## 3. Planification

Comment préparer et structurer le travail avant de développer un logiciel ?

Après avoir réalisé un état de l'art des logiciels existants, il est essentiel de planifier le développement du nouveau logiciel.

Du point de vue technique, cette étape repose sur trois éléments clés : définir l'architecture globale du logiciel, rédiger un plan de gestion logiciel (PGL) et choisir un environnement de développement adapté, tel qu'une forge logicielle.

Du point de vue juridique, cette étape repose sur le choix de la licence.

### 3.1 Design du logiciel

L'architecture globale d'un logiciel (en anglais "*design*") correspond à la vision d'ensemble de l'application qui sert de plan directeur avant le développement. Il permet de structurer le projet et d'assurer que tous les choix (fonctionnels, techniques et ergonomiques) sont cohérents.

En résumé, il couvre trois grands volets :

- **volet fonctionnel** : décrit ce que fait le logiciel et pour qui,
- **volet technique / architecture** : spécifie comment le logiciel sera construit, choix de technologies (langages, *frameworks*, bases de données),



- **volet interface et expérience utilisateur** : comment l'application se présente, comment l'application se vit et s'utilise.

### Pour aller plus loin

- Wikipedia. ISO/IEC 12207 [en ligne]. Disponible sur [https://en.wikipedia.org/wiki/ISO/IEC\\_12207](https://en.wikipedia.org/wiki/ISO/IEC_12207) [consulté le 3 février 2026].
- Wikipedia. High-level design [en ligne]. Disponible sur [https://en.wikipedia.org/wiki/High-level\\_design](https://en.wikipedia.org/wiki/High-level_design) [consulté le 3 février 2026].
- Wikipedia. Architecture logicielle [en ligne]. Disponible sur [https://fr.wikipedia.org/wiki/Architecture\\_logicielle](https://fr.wikipedia.org/wiki/Architecture_logicielle) [consulté le 3 février 2026].

## 3.2 Plan de gestion de logiciel (PGL)

Le plan de gestion de logiciel est un **document évolutif** qui permet aux chercheurs d'anticiper les questions liées au développement et à la valorisation des codes et logiciels. À travers des questions ciblées, le plan de gestion logiciel aide les chercheurs à :

- réfléchir en amont sur toutes les étapes du cycle de vie du logiciel,
- organiser le développement du logiciel,
- planifier sa diffusion et sa maintenance,
- identifier les utilisateurs cibles,
- évaluer l'impact du logiciel sur la recherche,
- assurer sa pérennité,
- garantir la conformité avec les principes FAIR pour les logiciels,
- vérifier le travail réalisé dans le cadre d'un projet financé (les financeurs et la communauté peuvent évaluer si les objectifs annoncés ont été atteints),
- réfléchir aux bonnes pratiques pour développer et diffuser un code ou un logiciel.

Dans le cadre d'un projet de recherche impliquant la création de codes, il est fortement recommandé d'assurer un lien entre le logiciel et les données correspondantes. Sur la [plateforme DMP OPIDoR](#), par exemple, lors de la rédaction d'un plan de gestion des données, il est possible de déclarer un code ou un logiciel comme produit de recherche. Pour ce produit spécifique, le chercheur répond à des questions dédiées, adaptées à la production de codes.

## Pour aller plus loin

- Chue Hong, N. P., Katz, D. S., Barker, M., Lamprecht, A.-L., Martinez, C., et al.. RDA FAIR4RS WG. FAIR Principles for Research Software (FAIR4RS Principles) (1.0). 2022. <https://doi.org/10.15497/RDA00068>
- Jackson, M. Software Management Plans. Software Sustainability Institute. 2016. <https://www.software.ac.uk/news/software-management-plans>
- Martinez-Ortiz, C., Martinez Lavanchy, P., Sesink, L., Olivier, B. G., Meakin, J., et al.. Practical guide to Software Management Plans (1.1). 2023. <https://doi.org/10.5281/zenodo.7589725>
- OPIDoR. Faciliter la mise en œuvre de bonnes pratiques de gestion et ouverture des codes sources et logiciels. 2025. <https://opidor.fr/faciliter-la-mise-en-oeuvre-de-bonnes-pratiques-de-gestion-et-douverture-des-codes-sources-et-logiciels/>
- Santangelo, M.G., Louvet, V., Chachay, S., Sadowska, J., Medves, M., et al.. Développement d'un formulaire machine-actionnable pour la gestion FAIR des logiciels de recherche dans les plans de gestion de projet. 2026. <https://hal-lara.archives-ouvertes.fr/hal-05536240v1>
- The Software Sustainability Institute. Checklist for a Software Management Plan (1.0). 2018. <https://doi.org/10.5281/zenodo.2159713>

### 3.3 Forge logicielle

La forge est l'environnement idéal pour **développer un code source répliquable**. Grâce à son **système de gestion de versions** et à ses **outils collaboratifs**, elle constitue un support indispensable dans la plupart des projets.

Une forge est une plateforme en ligne de gestion, partage et maintenance collaborative des codes sources. Elle intègre généralement :

- un système de gestion des versions (le plus utilisé étant [Git](#)),
- un éditeur en ligne,
- un gestionnaire de listes de discussion (listes, forums),
- un outil de suivi des bugs,
- une gestion des tâches,
- un gestionnaire de documentation (souvent sous forme de wiki),
- des outils de contrôle qualité,

- des outils d'intégration et de déploiement continus (CI/CD). L'intégration continue consiste à intégrer les correctifs à la branche centrale du code source. Le déploiement continu permet d'intégrer ces correctifs sans cesser le déploiement du site sous la forme d'une application ou d'un site web.

Dès la planification d'un projet, le choix de la forge doit se faire selon les besoins :

- certaines permettent un accès public au code source, d'autres non,
- certaines offrent des services supplémentaires (ex. génération de pages web avec GitLab Pages),
- certaines sont libres (GitLab), d'autres propriétaires (GitHub, propriété de Microsoft),
- elles diffèrent dans la gestion des interactions entre développeurs et utilisateurs (signalement de bugs, demandes d'évolutions, etc.).

Si les forges institutionnelles de la recherche sont souvent centrées sur leurs communautés (universités, laboratoires, écoles), il existe aussi des forges ouvertes à des projets plus larges, comme la [forge d'Huma-Num](#), qui accueille les développements dans le domaine des Sciences Humaines et Sociales.

### **Pour aller plus loin**

- Le Berre, D., Jeannas, J-Y., Di Cosmo, R., Pellegrini, F. Forges de l'Enseignement supérieur et de la Recherche : définition, usages, limitations rencontrées et analyse des besoins. Comité pour la science ouverte. 2023. [hal-04098702v7](#)
- CatOPIDoR. Forge logicielle [en ligne]. Disponible sur [https://cat.opidor.fr/index.php/Forge\\_logicielle](https://cat.opidor.fr/index.php/Forge_logicielle) [consulté le 3 février 2026].
- Gitlab Grenoble. Documentation de la plate-forme gricad-gitlab [en ligne]. Disponible sur <https://docs.gricad-pages.univ-grenoble-alpes.fr/help/> [consulté le 3 février 2026].

### **3.4 Choix d'une licence**

Le choix de la licence doit être fait au tout début du projet, idéalement avant même d'écrire la première ligne de code, car :

- il dépend souvent de la politique de l'institution ou du financeur (certains exigent des licences libres),
- il doit être cohérent avec les bibliothèques tierces utilisées (incompatibilité possible entre certaines licences),

- il permet d'éviter les blocages juridiques en fin de projet ou lors de la diffusion.

Pendant la planification, il est important d'identifier qui détient les droits patrimoniaux du logiciel et d'échanger avec le service de valorisation et les juristes pour un choix de licence le plus adapté.

Il existe différentes typologies de licence.

### 3.4.1 Licences permissives (sans copyleft)

Une licence permissive offre des possibilités de réutilisation, modification et distribution plus importantes que celles proposées par les autres licences. Les conditions de la licence initiale ne sont pas imposées et il est possible d'y ajouter de nouvelles restrictions.

- [The Unlicense](#)
- [Apache License](#)
- [MIT License](#)
- [Berkeley Source Distribution \(BSD\)](#)

### 3.4.2 Licences copyleft

Une licence copyleft donne la possibilité de réutiliser une œuvre libre tout en garantissant la préservation des libertés offertes initialement par la licence en cas de redistribution de l'œuvre, modifiée ou non.

Deux types existent avec :

**Copyleft faible** : la licence initiale reste, les ajouts peuvent recevoir une autre licence.

- [Licence publique générale GNU \(GPL\)](#)
- [Eclipse Public License \(EPL\)](#)
- [Mozilla Public License \(MPL\)](#)
- [Lesser licence publique générale limitée \(LGPL\)](#)
- [European Union Public Licence \(EUPL\)](#)

**Copyleft fort** : aussi appelée « contaminante », la licence initiale s'impose sur tout.

- [GNU General Public License \(GPL\)](#)
- [Affero GPL \(AGPL\)](#)

Ces licences sont dites **diffusives** (on trouve parfois dans l'industrie logicielle propriétaire le terme "contaminantes" ou "virales" pour qualifier péjorativement ces ressources qui font

partie de la famille des licences à copyleft fort) car elles obligent les concepteurs de logiciels qui utilisent des briques logicielles marquées par ces licences à remettre à la communauté sous la même licence les produits dérivés de ces briques.

### 3.4.3 Double licence

À des fins de valorisation, il est possible de prévoir une licence double (dual licence). Par défaut la licence sera restrictive (copyleft fort), mais il sera possible à des entreprises, moyennant compensation financière, de réaliser la brique logicielle dans le code dont elles sont propriétaires sans avoir à délivrer ce code comme le stipule les licences diffusives.

Les licences suivantes sont classées de la plus restrictive (GNU AGPLv3) à la plus libre (Unlicense) :

- [GNU AGPLv3](#)
- [GNU GPLv3](#)
- [GNU LGPLv3](#)
- [Mozilla Public License 2.0](#)
- [Apache License 2.0](#)
- [MIT License](#)
- [Boost Software License 1.0](#)
- [Unlicense](#)

### Pour aller plus loin

- Choosealicense.com. La table des correspondances des licences open source [en ligne]. Disponible sur <https://choosealicense.com/appendix/> [consulté le 3 février 2026].
- Choosealicense.com. Le logigramme interactif Choose an open source license [en ligne]. Disponible sur <https://choosealicense.com/> [consulté le 3 février 2026].
- Commission européenne. Le comparateur de licences [en ligne]. Disponible sur <https://interoperable-europe.ec.europa.eu/collection/eupl/solution/licensing-assistant/find-and-compare-software-licenses> [consulté le 3 février 2026].
- DATACC. Licences et propriété intellectuelle [en ligne]. Disponible sur <https://www.dataacc.org/partager/diffuser-ses-codes/licences-et-propriete-intellectuelle/> [consulté le 3 février 2026].

- SPDX. La liste des licences SPDX [en ligne]. Disponible sur <https://spdx.org/licenses/> [consulté le 3 février 2026].
- Tal, L. Licences open source : types et comparaison [en ligne]. Disponible sur <https://snyk.io/fr/articles/open-source-licenses/> [consulté le 4 mai 2026].

## 4. Développement

Cette phase transforme la conception en code fonctionnel. Quels choix techniques faire ? Comment bien organiser le travail collaboratif ? Comment avoir une documentation claire et des commentaires précis afin de garantir la compréhension et la réutilisation du code ?

La phase de développement est l'**étape la plus cruciale du cycle de vie du logiciel**. Au cours de cette étape, les idées issues des phases de planification et d'analyse des besoins sont mises en œuvre, testées et intégrées.

Il s'agit en théorie de :

- traduire les documents de conception détaillés en code,
- identifier et développer les unités fonctionnelles du logiciel.

Le contexte de développement oblige parfois les concepteurs à s'écarter de ce modèle idéal.

### 4.1 Les choix à faire

#### 4.1.1 Langage de programmation

Est-ce un langage compilé ou interprété, multiplateforme ou pas ?

#### 4.1.2 Modules ou classes

Les paradigmes de programmation sont fonctionnels, impératifs, orientés objet.

#### 4.1.3 Systèmes et utilisateurs ciblés

Les systèmes utilisés sont Windows, Linux, Unix, MacOS ...

Les utilisateurs ciblés sont les professionnels, ingénieurs, chercheurs, grand public

...

#### 4.1.4 Interfaces

S'agit-il d'une application web, d'une application autonome ou de systèmes embarqués ... ?

#### 4.2 Comment les collaborateurs vont-ils s'organiser pour développer le code ?

Il faut définir les principaux modes de collaboration s'il s'agit de projets de grande ampleur (qui s'occupe de faire les *merges* ou fusions, qui traite les problèmes remontés par les usagers sous formes d'issues, etc.).

Il faut également penser à la distribution des tâches.

#### 4.3 Documentation

Le code et les données de recherche doivent être clairement documentés, non seulement **pour les collaborateurs**, mais aussi pour d'**autres équipes** ou pour un **usage futur**. Il est important d'expliquer quelles fonctions traitent quels jeux de données. Pourquoi certains choix ont été faits. Quelle philosophie de développement guide le projet ou certaines de ses parties.

La documentation associée au code explique parfois comment certaines routines fonctionnent, mais pas forcément pourquoi ces routines ont été choisies.

La documentation passe notamment par des fichiers dédiés, tels que :

- fichier README : description générale du projet et instructions d'utilisation (fortement recommandé),
- fichier CONTRIBUTION / AUTEURS : modalités de contribution (fortement recommandé),
- fichier CHANGELOG : décrit les évolutions majeures (fortement recommandé),
- fichier CITATION : informations pour citer le logiciel (recommandé),
- fichier LICENCE : contrat juridique indiquant les conditions de réutilisation du code (fortement recommandé).

Enfin, il est essentiel de commenter le code directement, de préférence en anglais, afin d'en faciliter la lecture et la réutilisation par la communauté scientifique.

**Pour aller plus loin**

- Clark, A. What is the Software Development Life Cycle (SDLC)?. 2025.  
<https://cpoclub.com/product-development/software-development-life-cycle/>
- Michigan Technological University. System Development Lifecycle (SDLC) [en ligne]. Disponible sur <https://www.mtu.edu/it/security/policies-procedures-guidelines/information-security-program/system-development-lifecycle/> [consulté le 3 février 2026].

## 5. Tests

Comment tester un logiciel pour s'assurer qu'il fonctionne correctement, qu'il est fiable, compatible et performant ?

Les tests automatisés sont des scripts et des outils fournis avec le code pour vérifier automatiquement son bon fonctionnement. Ils appellent différentes fonctions et méthodes. Ils permettent de tester le code sans exécuter tout le programme.

Une suite de tests sert aussi de documentation, en indiquant comment les parties du code interagissent et doivent être utilisées.

Il existe un grand nombre de types de tests qui peuvent être effectués sur un code source. Les plus importants, dans le contexte d'un environnement de recherche, sont les tests unitaires et les tests d'intégration.

### 5.1 Tests unitaires

C'est le type de test le plus basique et le plus important à développer.

Son objectif est de **vérifier l'implémentation correcte des méthodes et fonctions uniques sur lesquelles un logiciel est construit.**

Il s'agit d'une suite de petits appels de fonctions mettant en évidence des cas particuliers et l'utilisation générale des fonctions implémentées.

La mise en œuvre d'un test unitaire aidera le développeur à :

- identifier les cas particuliers,
- assurer la stabilité et la justesse de l'implémentation,



- vérifier tout ce qui précède en une seule fois, sans avoir à appeler la fonction à plusieurs reprises avec tous les cas possibles.

## 5.2 Tests d'intégration

Dans le cas d'une application complexe, dans laquelle plusieurs modules et classes sont interconnectés pour fournir une fonctionnalité de niveau supérieur, les tests d'intégration visent à vérifier l'intégration correcte entre eux.

Prenons l'exemple d'une application qui doit effectuer les opérations suivantes :

- accepter un nom d'utilisateur et un mot de passe pour l'authentification,
- se connecter à une base de données pour recueillir des données,
- traiter les données collectées dans un cadre logiciel,
- enregistrer le résultat selon un certain format.

**Le but d'un test d'intégration est de répondre aux questions suivantes :**

- Que se passe-t-il si l'authentification échoue pour d'autres raisons qu'un nom d'utilisateur / mot de passe incorrect ?
- Que se passe-t-il si la base de données ne répond pas ?
- Quel est le résultat si les données collectées sont corrompues / incomplètes ?
- Que se passe-t-il si le format des résultats change ?

**Pour aller plus loin**

- Pittet, S. The different types of software testing. [consulté le 3 février 2026].

<https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing>

- Python

[UnitTest](#)

[PyTest](#)

[Coverage](#)

- R

[Testthat](#)

[tinytest](#)

- Javascript

[Jest](#)

- Java

[JUnit](#)

[TestNG](#)

- C

[cmocka](#)

[check](#)

## 6. Intégration continue

Comment tester et valider le code à chaque changement pour livrer un logiciel fiable et de qualité ?

L'intégration continue (de l'anglais : *Continuous integration, CI*), est un ensemble de pratiques utilisées en génie logiciel consistant à **vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée.**

L'objectif principal est de compiler, tester et déployer le code source des logiciels développés. Chaque modification de code dans une base de code est soumise à un ensemble d'étapes automatiques (*CI pipeline*) afin de s'assurer que :

- le nouveau code passe tous les tests,
- si l'un des tests échoue, le code n'est ni déployé ni accessible,
- le code disponible fonctionne toujours correctement.

Les pipelines d'automatisation des logiciels via des forges dédiées (par exemple, Github) facilitent la mise en place de la CI. Cependant, ces templates doivent être minutieusement adaptés au code pour être efficaces.

Un pipeline typique (qu'on peut définir comme un flux de travail, par exemple sur Github) se compose généralement des étapes suivantes :

- une *pull request* (proposition d'ajout au code existant) a été faite par l'un des contributeurs ;
- le code modifié est soumis à toutes les suites de tests disponibles ;
- si tous les tests sont réussis, le nouveau code est automatiquement fusionné dans la version actuelle du code.

## Pour aller plus loin

- [github actions](#)
- [github](#)
- [writing workflows](#)
- [redhat](#)

## 7. Maintenance

Comment garantir qu'un logiciel reste fiable, performant, et adapté aux besoins identifiés, au fil du temps ?

La maintenance des logiciels est le **processus de changement, de modification et de mise à jour des logiciels pour répondre aux besoins des usagers.**

Compte-tenu de la rapide évolution des logiciels, une maintenance est nécessaire ne serait-ce que pour garder le logiciel exécutable sur plusieurs années.

Différents types de maintenance peuvent être planifiés et exécutés, en fonction des exigences prévues lors de la phase d'élaboration des besoins et de la phase de planification.

Il existe quatre types de maintenance logicielle :

- **corrective** (résolution des problèmes),
- **adaptative** (mises à jour du système),
- **perfective** (amélioration des performances),
- **préventive** (prévention proactive des problèmes).

En général, dans le contexte d'un logiciel de recherche, il est plus courant de planifier et de réaliser uniquement les maintenances correctives et perfectives. En effet, les autres types de maintenance concernent souvent des bases de code plus vastes et plus complexes. Il est donc essentiel de **déterminer précisément le type de maintenance nécessaire pour chaque projet**, car il peut être difficile de prendre en charge l'ensemble des maintenances dans le cadre d'un projet de recherche.

## 7.1 Maintenance corrective

La maintenance corrective fait référence à un protocole de collaboration dans lequel les utilisateurs peuvent informer les développeurs d'un problème, d'un bug ou d'une erreur dans le logiciel. Ce qui déclenche ensuite une tâche de codage pour corriger ce bug et mettre à jour la version actuelle du logiciel.

Normalement, cela se fait via des outils collaboratifs tels que les forges (Gitlab, Codeberg, Forgejo, Github), qui permettent de signaler un problème rencontré lors de l'exécution du code (*issue*). Ce "ticket" sera pris en charge par un ou plusieurs développeurs afin d'obtenir une résolution et un correctif qui sera apporté au code source.

## 7.2 Maintenance perfective

La maintenance perfective est l'adoption de nouvelles structures de données, de nouveaux algorithmes ou de nouvelles solutions technologiques pour améliorer les performances, la sécurité ou, en général, les caractéristiques d'un logiciel donné. Le point de départ est l'utilisation extensive du logiciel qui conduit à la découverte de nouveaux besoins en termes de performance.

La contribution de la communauté des utilisateurs est cruciale pour cette phase.

### Pour aller plus loin

- Seriai, A-D. Évolution et restructuration du logiciel. 2015.  
<https://www.lirmm.fr/~seriai/uploads/Enseignement/cours.pdf>
- Thales. Qu'est-ce qu'un processus de maintenance logicielle ? 4 types de maintenance logicielle [en ligne]. Disponible sur  
<https://cpl.thalesgroup.com/fr/software-monetization/four-types-of-software-maintenance> [consulté le 3 février 2026].

## 8. Ouverture du code

Les codes de la recherche sont une production scientifique à part entière.

Comment les faire connaître à la communauté ? Quelles bonnes pratiques peuvent être développées ?

L'ouverture d'un logiciel de recherche fait référence à sa disponibilité et accessibilité pour d'autres utilisateurs ou équipes, souvent sous la forme d'un logiciel open source.

Elle permet :

- le partage des résultats et des outils avec la communauté scientifique,
- la collaboration et la contribution d'autres chercheurs ou développeurs,
- la transparence et la reproductibilité des expériences scientifiques,
- l'amélioration continue grâce aux retours et aux contributions externes.

L'ouverture implique également une documentation claire, une licence adaptée et des procédures pour contribuer ou signaler des bugs, afin que le logiciel puisse être utilisé et étendu de manière fiable.

Selon le site [code.gouv.fr](https://code.gouv.fr) quatre degrés d'ouverture pour les codes sources sont pris en considération :

- **contributif** : les contributions sont recherchées et gérées activement, tous les codes de recherche n'ont pas vocation à recueillir des contributions,
- **ouvert** : les contributions sont gérées mais pas recherchées,
- **publié** : les contributions extérieures ne sont pas traitées,
- **non-communicable** : le code source n'est pas ouvert.

Bien sûr, cela est fortement réglementé par la licence logicielle choisie et le guide pratique de contribution. Les deux doivent être définis dès les premières étapes de la planification et du développement.

### Pour aller plus loin

- DATAACC. Bonnes pratiques de développement [en ligne]. Disponible sur <https://www.dataacc.org/organiser/bonnes-pratiques-de-developpement/> [consulté le 3 février 2026].
- Data.gouv.fr. Codes sources du secteur public : lesquels ouvrir, pourquoi et comment ? [en ligne]. Disponible sur [consulté le 3 février 2026].

## 9. Contributions

Dans quelles situations le public peut-il contribuer à un code ?

Comment permettre aux membres de la communauté de contribuer au code ?

Dès qu'un logiciel de recherche devient connu, d'autres personnes peuvent vouloir y contribuer : retours, nouvelles fonctionnalités, etc.

La création d'une version dérivée (*fork*) du dépôt permet de tester ces contributions tout en conservant la version originale. Les contributions se font souvent via des *pull requests* (demandes de tirage) qui sont révisées par les administrateurs du projet.

Si elles sont acceptées, elles sont intégrées au code après tests pour vérifier qu'elles fonctionnent correctement et n'altèrent pas le reste du logiciel.

Les ajouts importants sont mentionnés dans le CHANGELOG, fichier qui recense toutes les modifications importantes apportées à un logiciel depuis sa première version.

Les auteurs sont référencés dans le Citation File Format (CFF), fichier qui permet de citer correctement un logiciel ou un projet open source.

La gestion des contributions doit être documentée en précisant :

- le type de contributions attendues (bugs, nouvelles fonctionnalités, etc.),
- le protocole de soumission,
- les droits des participants,
- les critères d'acceptation.

Ces informations peuvent être renseignées dans une documentation spécifique à destination des contributeurs.

Une documentation bien fournie facilite grandement la proposition de contributions autant que la qualité de celles-ci. Comment en effet pourrait-on proposer des améliorations à un code source dont on ne comprend pas bien :

- comment il fonctionne,
- pourquoi certaines fonctions ont été choisies et pas d'autres qui auraient paru au moins aussi pertinentes.

Cette documentation facilitera une meilleure constitution de la communauté autour du logiciel.

## **10. Rapport de bug**

Dès que le logiciel est ouvert à la communauté depuis la forge, il est susceptible d'être testé et réutilisé par des personnes extérieures au projet. Comment être en mesure de gérer les bugs qui seront rapportés à partir de ce moment ?

Les forges disposent généralement d'un système d'issues qui permettent aux usagers du code de déposer des demandes de résolution de bugs constatés ou bien des demandes d'évolution du code source.

### **10.1 Recueillir des rapports de bugs complets**

Les utilisateurs du code source ne sont pas toujours très conscients des informations utiles à transmettre au moment de rapporter un bug, informations dont le développeur a besoin pour reproduire le problème et le corriger. Pour guider l'utilisateur dans son rapport de bug, chaque forge propose des lignes directrices différentes, dont le style peut différer mais dont la substance est la même.

### **10.2 Trier et assigner les bugs**

Le tri, en fonction de la sévérité et l'affectation des bugs est l'une des tâches du gestionnaire du projet logiciel, par exemple, le développeur principal. Il doit fournir des instructions claires comme rechercher les bugs déjà signalés avant d'en déposer un nouveau.

S'il y a plusieurs développeurs impliqués, il faut ensuite déterminer, en fonction des contributions de chacun, qui s'occupera de corriger quoi. Les forges intègrent un certain nombre d'outils qui peuvent être activés pour aider à trier, classifier et suivre la résolution de ces bugs en qualifiant ces différentes étapes. C'est le cas par exemple de Bugzilla qui peut être utilisé à travers Gitlab.

Si les corrections développées passent la suite de tests, alors la fonction *merge* (fusion) de la forge permettra de les ajouter à la version stable.

Il est important de noter que si la plupart des bugs sont le résultat d'un code mal développé, certains d'entre eux peuvent être la conséquence de mauvais choix de conception lors de la phase de planification. Il est essentiel de conserver une vision claire de ce qui a été conçu et planifié avant de commencer à travailler sur une solution.

### Pour aller plus loin

- OpenSC. How to write a good bug report?. 2024.  
<https://github.com/OpenSC/OpenSC/wiki/How-to-write-a-good-bug-report>

## 11. Publication et Archivage

Comment publier le code sur une forge logicielle ou dans un *software paper* pour le rendre accessible à la communauté ? Comment l'archiver pour assurer sa pérennité ?

Publier le code source est une étape clé d'un projet : cela permet de **l'ouvrir à la communauté, de recueillir des retours et de documenter un résultat scientifique**. Le rendre public sur une forge assure sa visibilité, et, lorsqu'il est suffisamment robuste et réutilisable, il peut faire l'objet d'une publication dédiée, appelée *software paper*. En revanche, seul l'archivage garantit sa pérennité.

### 11.1 Publier depuis la forge

Publier un logiciel depuis une forge logicielle permet de partager facilement le code avec la communauté ou ses collaborateurs, suivre les modifications grâce au versionnage, et gérer les contributions. Les forges offrent aussi la possibilité de créer des *releases* (versions), d'automatiser les tests et de documenter le projet de manière centralisée.

Une étape importante est de **bien préparer le logiciel** :

- **organiser le projet** : assurer que le code est propre et structuré,



- **ajouter de la documentation** : inclure un README expliquant ce que fait le logiciel et comment l'installer / utiliser,
- **définir une licence** : ajouter un fichier LICENSE pour préciser les droits d'utilisation,
- **gérer les versions** : utiliser un schéma de versionnage comme Semantic Versioning.

### Pour aller plus loin

- [Semantic Versioning](#)
- [GitLab generic packages repository](#)

## 11.2 Publier un software paper

Un software paper est une **publication revue par les pairs dont l'objectif est de présenter un logiciel à la communauté scientifique**. Le logiciel décrit est généralement accessible sous une licence permettant de le réutiliser. L'article doit faire un lien vers la version du logiciel disponible au moment de la publication de l'article et peut aussi faire un lien vers les versions plus récentes.

Il existe essentiellement deux types de software paper en fonction des revues :

- **un article autonome** qui décrit uniquement le logiciel, généralement dans un format plus court qu'un article écrit sur une étude de recherche traditionnelle (cf. cet article décrivant une [nouvelle Suite pour Matlab](#)),
- **un article (plus traditionnel) qui décrit une question de recherche originale incluant le développement d'un nouveau logiciel** (voir par exemple cet article paru sur [BMC Neurosciences](#)).

### 11.2.1 Avantages de la publication d'un software paper

- Faire connaître un logiciel
- Démontrer la robustesse et la qualité d'un logiciel
- Valoriser le travail des développeurs
- Offrir de la transparence sur le processus de développement du logiciel et de son code source

- Faciliter la réutilisation et la citation d'un logiciel dans le cadre de recherches ultérieures

### 11.2.2 Où publier un software paper ?

- Le tableau [Data & Code journals](#) de l'EPFL
- La liste [In wich journals should I publish my software?](#) du Software Sustainability Institute
- La base d'information [Où publier ?](#) du CIRAD
- Dans les revues et actes de conférences qui se concentrent sur les domaines de la recherche informatique ([ACM Transactions on Mathematical Software](#) ; [BMC Bioinformatics](#) ; [Computing in Science & Engineering](#) ; [Journal of Software Engineering Research and Development \(opens in a new tab\)](#) ; [SIAM Journal on Scientific Computing](#) ; [SoftwareX](#))
- Dans un software journal ([Journal of Open Research Software](#) ; [Journal of Open Source Software](#) ; [Image Processing On Line \(IPOL Journal\)](#))
- Les plateformes de publication de travaux de recherche ([Open Research Europe](#))

### Pour aller plus loin

- Institut Pasteur. Les software papers. 2024.  
<https://openscience.pasteur.fr/2024/02/26/les-software-papers/>
- Romano, J.D., Moore, J.H. Ten simple rules for writing a paper about scientific software. 2020.  
<https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1008390>

## 11.3 Archiver

Reproduire une expérience, c'est notamment **être en mesure de répliquer le logiciel ayant servi au traitement des données**. Pour cela, le logiciel doit rester accessible sur le long terme. Or, la temporalité des forges logicielles n'est pas toujours celle de la conservation : elles peuvent disparaître si leur hébergeur cesse de les maintenir. Pour préserver durablement un logiciel, il faut donc recourir à une

archive adaptée à ce type d'artefact. La fondation Software Heritage propose précisément ce service.

### 11.3.1 Présentation de Software Heritage

[Software Heritage](#) est un organisme à but non lucratif qui s'est donné comme mission de sauvegarder le code source et le patrimoine informatique des institutions de recherche. Cette archive moissonne les entrepôts de code des forges grand public comme Github. Toutefois, si le chercheur veut s'assurer que son code soit bien archivé, il lui faut pousser son code vers l'archive, opération qui peut être renouvelée autant de fois que nécessaire.

À chaque version archivée du code est attribué un identifiant unique et persistant différent. Cet identifiant est connu sous le nom de **SWHID** ([SoftWare Hash Identifiers](#)). Grâce à cet identifiant, il sera possible de citer une version d'un code source, voire quelques lignes de cette version dans une publication. Cet identifiant pourra en outre être renseigné dans les métadonnées d'autres artefacts de la recherche comme la publication sur HAL ou les données présentes dans l'entrepôt de données.

### 11.3.2 Comment archiver le code dans Software Heritage depuis la forge ?

Software Heritage met à disposition pour Mozilla Firefox et des navigateurs chromium-like un plugin qui permet de pousser le code de la forge où il se trouve (github, gitlab, codeberg, etc.) vers l'archive.

On peut bien entendu réaliser cette opération depuis Software Heritage en renseignant le dossier (repository) de la forge où se trouve le code et les fichiers associés.

#### Pour aller plus loin

- Software Heritage. How to archive and reference your code [en ligne]. Disponible sur <https://www.softwareheritage.org/how-to-archive-reference-code/> [consulté le 9 février 2026].

## 11.4 Référencer

Déposer un logiciel sur la plateforme HAL consiste à créer une notice dédiée décrivant précisément le projet : nom du logiciel, auteurs, résumé, mots-clés, ainsi que le lien vers le code source (par exemple sur GitHub ou GitLab).

Il est recommandé d'ajouter une documentation, une licence d'utilisation et, si possible, une version archivée associée à un identifiant pérenne (comme le SWHID).

Le dépôt de logiciels dans **HAL** permet notamment leur **citabilité**, tandis que l'**archivage du code** est pris en charge par l'infrastructure **Software Heritage**. **HAL et Software Heritage proposent ainsi des services complémentaires.**

Pour enrichir automatiquement la description du logiciel, HAL peut exploiter les métadonnées présentes dans le fichier CodeMeta, facilitant la saisie et améliorant la qualité des informations.

On parle parfois de CODEMETA.Json parce qu'aujourd'hui, le javascript est le langage le plus utilisé pour l'écrire.

[Des formulaires existent](#) pour remplir ce fichier sans avoir à s'approprier sa syntaxe.

Si le logiciel est archivé dans Software Heritage et rendu visible dans HAL au moyen du CODEMETA, il pourra être indexé dans le catalogue des logiciels libres de la recherche académique. Cela lui donnera davantage de visibilité et lui permettra d'être réutilisé. Il pourra également faire l'objet de propositions d'améliorations.

Pour s'informer sur le processus complet du dépôt et sur la rédaction du fichier des métadonnées CODEMETA, se reporter à la présentation Gruenpeter, Sadowska, 2022 dans la bibliographie.

### Pour aller plus loin

- Gruenpeter, M., Sadowska, J., Nivault, E., Monteil, A., 2022. Create software deposit in HAL [en ligne]. Disponible sur <https://inria.hal.science/hal-01872189v2> [consulté le 7 avril 2026].

- HAL. Déposer le code source d'un logiciel [en ligne]. Disponible sur <https://doc.hal.science/deposer/deposer-le-code-source/> [consulté le 9 février 2026].

Lors de futures publications, on peut citer les logiciels qui ont servi de base pour le travail au même titre que les articles, ouvrages, etc. Lorsque l'on citera ces logiciels, il faudra indiquer : le ou les auteurs, le titre du logiciel, le lieu de publication, la date de publication associée à la version de logiciel, l'identifiant, l'URL permettant d'accéder au logiciel, la version.

Le package biblatex-software disponible sur CTAN sait gérer les entrées bibliographiques pour le logiciel. Celles-ci peuvent aussi être importées dans Zotero grâce à l'extension BetterBibLatex.

### **Pour aller plus loin**

- Smith, A.M., Katz, D.S., Niemeyer, K.E., 2016. Software citation principles [en ligne]. Disponible sur <https://peerj.com/articles/cs-86/> [consulté le 7 avril 2026].