# quickDoc

An ergonomic lightweight markup language with mnemonic inspirations for writing all kind of documents

Cyprien PIERRE

2025-10-24

**Résumé**

Rédiger le résumé

**Mots-clés :** Mot clé 1, Mot clé 2

## 1 Introduction

Le présent document définit **quickDoc**, un nouveau langage de balisage léger conçu pour remplacer l'écosystème vieillissant TEX/LATEX/BibLaTeX et les formats markdown actuels. quickDoc vise à unifier les avantages de ces systèmes tout en répondant à leurs limitations techniques, sémantiques et ergonomiques. Il est destiné à la rédaction de documents scientifiques, littéraires, de documentation technique et de publications, avec des sorties PDF et HTML5 conformes aux normes d'accessibilité WCAG niveau AAA.

**quickDoc** permet de stocker et d'exprimer :

— du contenu textuel formaté (sections, emphase, tableaux, etc.),
— des données structurées (métadonnées, configurations, tables de données typées),
— du code exécutable intégré (avec affichage du résultat dans le document),
— des formules mathématiques (évaluables ou non par un moteur de calcul),
— des références bibliographiques standard (interopérant avec CSL/BibTeX),
— des figures, diagrammes, graphiques et annotations avec leur sémantique.

Ce document est rédigé sous la forme d'une spécification normative de type ISO/RFC. Il énonce la terminologie, le modèle formel (syntaxe et grammaire), les exigences fonctionnelles et non-fonctionnelles, et les règles d'accessibilité et d'interopérabilité du langage quickDoc. Les mots "doit" et "ne doit pas" indiquent des exigences obligatoires, tandis que "devrait" indique une recommandation.

# 2 Etat de l'art

La production de documents formatés suit deux approches dominantes. La première repose sur des traitements de texte WYSIWYG comme MS Word, Google Docs ou OnlyOffice, qui exposent graphiquement les opérations de mise en forme mais atteignent vite leurs limites pour des documents exécutables, c'est-à-dire des textes où des éléments sont évalués puis remplacés par leurs résultats, comme du code générant un schéma ou un diagramme [a] [1, 2, 3]. La seconde approche consiste à écrire le contenu et son balisage dans un éditeur texte (Neovim, Emacs, iA Writer, etc.) puis à confier le rendu à un outillage adapté, ce qui place un langage de balisage au cœur du flux de rédaction, avec des convertisseurs assurant la transformation vers HTML, PDF ou EPUB [4, 5]. Dans ce cadre, on distingue des langages formels orientés structure et diffusion comme TEX ou HTML, et des langages de balisage léger (Lightweight Markup Language (LML)) tels que Markdown, Org-mode, AsciiDoc ou reStructuredText, préférés pour leur lisibilité en clair et leur intégration dans les pratiques « docs-as-code » [6, 7, 8, 9].

TEX s'impose historiquement pour la composition de haute qualité via des macros qui sous-tendent LATEX, ConTeXt, Texinfo et OpTeX, avec un écosystème de packages couvrant figures, bibliographies et disciplines spécialisées, mais au prix d'une infrastructure lourde, de choix techniques nombreux et de diagnostics d'erreurs difficiles sur de longs documents [10, 11, 12]. Des assistants comme LTeX+ et Writefull améliorent la qualité rédactionnelle et la conformité, sans toutefois résoudre les contraintes structurelles de la toolchain [13, 14]. La question de l'accessibilité demeure critique : la production de PDF balisés et conformes aux exigences WCAG reste incertaine sans post-traitements spécifiques, en particulier pour les mathématiques, ce qui rend nécessaire l'intégration en amont de métadonnées et structures adaptées pour des sorties réellement utilisables par les technologies d'assistance [15, 16, 17, 18].

Des systèmes récents cherchent à moderniser l'expérience de composition en combinant syntaxe compacte, prévisualisation rapide, contrôle programmatique de la mise en forme et messages d'erreur explicites, illustrant une direction de conception dont quickDoc peut s'inspirer pour viser une réactivité proche du temps réel tout en évitant les multiples passes et l'opacité des chaînes TEX traditionnelles [10, 12]. D'autres approches, comme Scribble dans l'écosystème Racket, montrent l'intérêt d'une documentation programmable où le document est un programme générant du contenu, ce qui offre une extensibilité considérable mais reste fortement couplé à l'environnement hôte et peu accessible hors de celui-ci ; l'objectif pour quickDoc est de conserver l'extensibilité sans imposer la lourdeur d'un langage généraliste aux usages courants [16].

Côté LML, Markdown demeure le plus populaire grâce à une syntaxe minimale et lisible, mais son périmètre limité a provoqué la prolifération de dialectes hétérogènes selon les plateformes, malgré les efforts de normalisation de CommonMark et la documentation de variantes comme GFM et le registre IANA ; cette diversité impose aux auteurs d'apprendre des sous-ensembles distincts et nuit à la portabilité inter-outils [6, 19, 20, 21]. Org-mode, très intégré à Emacs, apporte une puissance sémantique et outillée supérieure pour la planification, les tableaux, l'exécution de code avec résultats reproductibles et l'export multi-formats, mais sa dépendance à Emacs limite sa portabilité et son adoption hors de cet environnement [7]. AsciiDoc, aujourd'hui sous

a. **Document exécutable :** Désigne un document dont des éléments sont exécutés (p. ex. code) et remplacés par le produit de l'exécution (p. ex. schéma ou diagrammes).

gouvernance Eclipse, propose une couverture quasi exhaustive des besoins de documentation avec admonitions, inclusions, variables et macros, soutenue par une spécification et un TCK en cours de formalisation, ce qui en fait un choix robuste pour la documentation logicielle et les manuels, au prix d'une courbe d'apprentissage plus exigeante que Markdown [8, 22, 23, 24]. reStructuredText, conçu pour l'écosystème Python et Sphinx, offre des directives extensibles, des références croisées et des structures de haut niveau comparables à AsciiDoc, mais sa verbosité et ses conventions strictes le réservent souvent à son toolchain d'origine [9].

# 3 Limites et opportunités

L'émergence d'un besoin unifié naît des limites techniques, sémantiques et ergonomiques observées dans les systèmes existants et des exigences nouvelles liées aux flux « spec-driven » exploités avec des LLM, qui exigent des spécifications stables, testables et outillées dès le dépôt Git [25, 26, 27, 28]. Depuis la création de Markdown, les LML se sont diffusés dans les dépôts, wikis, chaînes de documentation et blogs avec la promesse d'un texte lisible et convertible, mais la prolifération de variantes incompatibles et l'absence initiale de norme robuste ont généré des ambiguïtés et des coûts de portabilité entre outils [6, 19, 20, 21]. Les utilisateurs pointent l'insuffisance des fonctionnalités natives pour les tableaux complexes, les références croisées et l'inclusion de fichiers, qui forcent à mélanger HTML et extensions, tandis que des alternatives plus riches comme AsciiDoc, reStructuredText et Org-mode demeurent surtout adoptées par des communautés spécialisées [7, 8, 9, 22].

Les critiques récurrentes s'articulent autour de la non-standardisation de Markdown et de ses dialectes, qui obligent à composer avec plusieurs moteurs de rendu et alourdissent la charge cognitive, alors que des efforts de normalisation et de test ont montré leur efficacité quand ils sont accompagnés d'une spécification et d'un TCK publics [6, 20, 23, 29]. Le manque de fonctions intégrées pour les notes, les références, les inclusions et les tableaux conduit à des pipelines hétérogènes, tandis que des langages orientés publication apportent admonitions, variables, macros et inclusions, au prix d'une syntaxe plus dense et d'outils dédiés (Asciidoctor, Antora, Sphinx, Emacs) [8, 9, 24, 30]. La courbe d'apprentissage reste paradoxale : Markdown est privilégié pour sa lisibilité sur de petits documents, mais AsciiDoc et reST abaissent l'effort cognitif sur de larges corpus grâce à des structures et blocs dédiés [9, 22, 27].

Les points positifs et attentes se concentrent sur la lisibilité en clair, l'adoption large et la portabilité multiformat, mais les utilisateurs veulent une combinaison de simplicité Markdown et de richesse d'AsciiDoc/reST, notamment admonitions, tableaux, variables, macros et système d'inclusion, intégrés sans HTML brut [8, 22, 24, 31]. L'extensibilité pensée pour la publication et la conversion vers HTML, EPUB, PDF ou DocBook est appréciée, surtout quand elle s'adosse à un convertisseur pivot documenté et scriptable [4, 5, 9]. La lisibilité en texte brut reste un critère central pour la collaboration, la relecture et l'onboarding de contributeurs hétérogènes [6, 8].

Les recommandations qui en découlent convergent vers un noyau syntaxique minimal complété par des modules optionnels pour monter en puissance, afin de préserver la simplicité tout en couvrant admonitions, références croisées, imports, variables et macros, avec une syntaxe concise et uniforme pour tableaux, listes et blocs dédiés [9, 22, 24, 31]. L'architecture devrait

être modulaire via plugins et blocs personnalisés, garantir des conversions fidèles vers HTML, PDF et EPUB, harmoniser la syntaxe des liens et employer des marqueurs explicites pour les opérations ambiguës, tout en conservant la lisibilité en clair [4, 6, 20]. La discipline de standardisation, soutenue par tests d'acceptation et TCK en intégration continue, est identifiée comme levier majeur pour limiter la dérive des variantes et sécuriser l'écosystème outillé, y compris dans des flux spec-driven avec LLM [23, 25, 29].

Les choix utilisateurs reflètent un arbitrage coûts-bénéfices entre « simplicité et adoption » et « puissance et structure » : Markdown est choisi par défaut pour sa popularité et ses intégrations plateformes, alors qu'AsciiDoc est mobilisé pour les manuels et publications structurées malgré l'installation d'outils spécifiques ; la décision dépend de la courbe d'apprentissage, de la prévisualisation, de la portabilité, de la compatibilité avec les workflows et de la collaboration [8, 26, 27]. Ce constat alimente un paradoxe : le langage le plus populaire n'est pas le mieux adapté à la documentation technique avancée, les freins à l'adoption de langages plus robustes étant surtout organisationnels et cognitifs, d'où l'intérêt d'un noyau minimal modulaire et d'une normalisation explicite des extensions [6, 9, 20].

Sur le plan cognitif, des marqueurs visibles et des structures uniformes réduisent la charge mentale et facilitent l'adoption, tandis que les admonitions et blocs dédiés améliorent la signalétique de l'information critique ; des mécanismes de portabilité tels que variables, macros et includes simplifient la maintenance et la réutilisation à l'échelle des dépôts [4, 8, 24]. La recherche future devrait analyser des corpus plus larges et des pratiques d'entreprises en « docs-as-code », quantifier la charge cognitive selon les LML chez novices et experts, et mesurer l'effet de la standardisation sur l'efficacité d'équipe et la qualité documentaire [26, 32, 33].

Les limites techniques et fonctionnelles appellent une spécification unique et stable qui empêche la dérive dialectale en définissant rigoureusement chaque fonctionnalité et en associant un banc de tests public, à l'image des démarches CommonMark, GFM et AsciiDoc-Lang [6, 20, 23]. Les chaînes TEX/LATEX souffrent de compilations multi-passes coûteuses sur gros documents, et un LML moderne devrait viser une compilation efficace et incrémentale avec des diagnostics clairs, dans un esprit de réactivité proche des pipelines continus observés dans les toolchains contemporaines [10, 11, 12]. L'accessibilité reste insuffisante par défaut : produire des sorties conformes WCAG et PDF/UA demande aujourd'hui des interventions spécifiques, d'où la nécessité d'intégrer nativement un balisage, des structures et des métadonnées A11Y-compatibles de bout en bout [15, 16, 18]. Le support de domaines comme mathématiques, chimie ou musique doit équilibrer expressivité et simplicité ; une intégration propre des formules, bibliographies et figures, avec éventuelle délégation calculatoire contrôlée, évite la dépendance à des modules externes ad hoc [34, 35, 36]. Enfin, un LML fondé sur une grammaire formelle déclarative et un Abstract Syntaxe Tree (AST) intermédiaire facilite parseurs robustes, linting, conversions et validations, comme l'illustrent les travaux sur grammaires formelles et MCSG pour langages « next-gen » [29, 37, 38].

Les limites sémantiques découlent du mélange fréquent entre présentation et contenu en LATEX et du sous-balisage sémantique en Markdown ; un LML devrait proposer des rôles et styles nommés pour annoter code, noms propres, termes de glossaire, équations et objets multimédias, ce qui est cohérent avec les exigences d'accessibilité et de restitution assistée [9, 16, 24]. Les structures de haut niveau (figures avec légende et texte alternatif, tableaux titrés, références croisées d'objets numérotés) doivent exister dans un modèle unifié, au-delà des directives

propres à chaque outil [8, 9]. Un mécanisme normé de métadonnées et de données embarquées, compatible avec les front-matter usuels et exportable en JSON, favorise interopérabilité et réutilisation [4, 5]. La possibilité de marquer des expressions comme calculables, ou d'introduire des champs interactifs, ouvre la voie à des documents actifs côté HTML tout en restant statiques côté archivage [34, 36].

Les limites ergonomiques plaident pour éviter la « soupe syntaxique » et imposent une seule forme par intention, une syntaxe auto-explicite et lisible en clair, sans dépendance à un environnement spécialisé, tout en offrant des éditeurs enrichis facultatifs et des messages d'erreur précis [7, 17, 38]. Des diagnostics au moment de la rédaction, une documentation normative accompagnée de tutoriels et d'exemples, et des validateurs intégrés à l'outillage CI renforcent l'adoption et la qualité globale [4, 6, 23].

En synthèse, ces constats justifient la création de quickDoc : un langage à noyau minimal, modulaire, formellement spécifié et testé, lisible en clair, extensible par profils contrôlés, couvrant nativement les blocs structurants, l'accessibilité, la sémantique et la portabilité multiformat, et aligné avec des workflows spec-driven compatibles LLM [6, 16, 23, 24, 25].

# 4 Exigences du langage

Sur la base des analyses précédentes, le présent chapitre formalise les exigences du langage quickDoc. Chaque exigence (`REQ.NN`) exprime un objectif général et chaque spécification (`SPEC.NN.MM`) détaille les conditions de satisfaction correspondantes. Les tests et règles associées seront dérivés de ces spécifications.

## 4.1 Fonctionnalités fondamentales

**REQ.01** quickDoc doit offrir un ensemble complet de mises en forme textuelles dans une syntaxe unifiée et non ambiguë.

- **SPEC.01.01** Les éléments de mise en forme comprennent au minimum : titres hiérarchiques, emphases, gras, italique, souligné, barré, exposant, indice, surlignage et commentaires invisibles.
- **SPEC.01.02** Une seule syntaxe doit exister pour chaque opération donnée afin d'éviter la multiplicité de notations observée dans Markdown et ses variantes [6, 20, 21].
- **SPEC.01.03** Les marqueurs doivent être explicites, lisibles et cohérents sur l'ensemble du langage afin de limiter la charge cognitive de l'utilisateur [22, 38].

**REQ.02** quickDoc doit intégrer les éléments structurés nécessaires à la production de documents techniques, scientifiques et éditoriaux.

- **SPEC.02.01** Figures et images doivent pouvoir être insérées avec légende, texte alternatif et ancrage logique dans la structure du document.
- **SPEC.02.02** Les tableaux doivent comporter titre, légende, en-têtes explicites et être navigables par les technologies d'assistance [9].
- **SPEC.02.03** Notes de bas de page et remarques en marge doivent être gérées nativement, avec possibilité de placement configurable.

**SPEC.02.04** Citations bibliographiques et références croisées internes (sections, figures, équations, listings) doivent être automatiques et maintenues à jour lors de la compilation.

**SPEC.02.05** Les admonitions (avertissements, exemples, remarques) doivent être intégrées dans le cœur du langage [24].

**REQ.03** quickDoc doit permettre l'inclusion et la gestion de données structurées à l'intérieur d'un document.

**SPEC.03.01** Les métadonnées documentaires (titre, auteur, date, licence, résumé, mots-clés) doivent être exprimées dans un bloc standard de type front-matter.

**SPEC.03.02** Les données applicatives (tableaux, variables, paramètres) doivent être déclarables, accessibles et réutilisables dans le corps du document [4].

**SPEC.03.03** L'export JSON ou YAML des métadonnées et des structures de document doit être garanti pour interopérabilité avec d'autres systèmes (par ex. doc-as-code, pipelines CI/CD).

**REQ.04** quickDoc doit permettre la production de documents reproductibles intégrant code et résultats.

**SPEC.04.01** L'inclusion de blocs de code (Python, R, Julia, Bash, etc.) doit être possible avec syntaxe claire.

**SPEC.04.02** Un paramètre doit définir si ces blocs sont exécutés ou non lors de la compilation.

**SPEC.04.03** Les résultats d'exécution doivent être automatiquement insérés au bon emplacement (texte, tableau, graphique).

**SPEC.04.04** Le comportement doit être aligné avec les principes du literate programming et des workflows reproductibles [1, 2].

**REQ.05** quickDoc doit gérer nativement les expressions mathématiques et leur évaluation.

**SPEC.05.01** Les formules doivent être exprimées dans une syntaxe textuelle claire, compatible avec LaTeX ou AsciiMath.

**SPEC.05.02** Certaines formules peuvent être marquées comme « calculables » et évaluées via un solveur symbolique externe (CAS).

**SPEC.05.03** Les graphiques ou courbes issus d'une formule doivent pouvoir être insérés automatiquement.

**SPEC.05.04** Les résultats numériques doivent être formatés selon les règles de locale et de typographie scientifique [35, 36].

**REQ.06** quickDoc doit intégrer une gestion bibliographique standardisée.

**SPEC.06.01** Le système de citation doit être compatible avec CSL-JSON, BibTeX et BibLaTeX.

**SPEC.06.02** Le style bibliographique doit être sélectionnable parmi la collection CSL existante.

**SPEC.06.03** Les bibliographies doivent être générées automatiquement et réactualisées à la compilation [4, 33].

**REQ.07** quickDoc doit disposer d'un mécanisme d'extension contrôlé.

**SPEC.07.01** L'utilisateur peut définir des macros nommées pour automatiser motifs et constructions récurrentes.

**SPEC.07.02** Les macros ne doivent pas altérer la grammaire de base ni rompre la compatibilité inter-documents.

**SPEC.07.03** Le système d'extension doit être auditable et documenté (manifestes, dépendances, versionnage) [23, 25].

## 4.2 Structure, sémantique et modélisation

**REQ.08** quickDoc doit disposer d'une grammaire formelle publiée et stable.

**SPEC.08.01** La syntaxe doit être décrite en EBNF et les structures de données documentées via schémas JSON.

**SPEC.08.02** Tout changement de version doit maintenir la rétrocompatibilité ou fournir un mécanisme de migration automatique [29, 37].

**SPEC.08.03** Un AST (Abstract Syntax Tree) formel doit être défini pour permettre les transformations, linting et conversions vers d'autres formats [38].

**REQ.09** quickDoc doit permettre une annotation sémantique riche du contenu.

**SPEC.09.01** L'utilisateur doit pouvoir marquer les entités textuelles par rôle : terme technique, code, nom propre, abréviation, acronyme, etc.

**SPEC.09.02** Les formules, figures, tableaux et citations doivent être typées dans le modèle de données (ex. figure scientifique, graphique statistique).

**SPEC.09.03** Les annotations doivent être exploitables pour l'accessibilité, la génération de métadonnées et la navigation contextuelle [16, 39].

**REQ.10** quickDoc doit permettre la modularisation et la composition documentaire.

**SPEC.10.01** Le langage doit inclure des directives d'inclusion ou d'import pour agréger des fichiers partiels.

**SPEC.10.02** La compilation doit reconstituer l'arborescence logique du document global.

**SPEC.10.03** Les inclusions doivent pouvoir être paramétrées (chemins relatifs, variables d'environnement).

**REQ.11** quickDoc doit garantir la cohérence hiérarchique et logique des structures.

**SPEC.11.01** Les titres doivent suivre une hiérarchie continue sans saut de niveau.

**SPEC.11.02** Les listes, tableaux et figures doivent être insérables dans n'importe quelle section sans altérer la structure.

**SPEC.11.03** Les références internes doivent être résolues de manière déterministe et vérifiées à la compilation.

## 4.3 Accessibilité, ergonomie et cognition

**REQ.12** quickDoc doit garantir la production de documents accessibles sans post-traitement manuel.

**SPEC.12.01** Les sorties PDF doivent être conformes PDF/UA et les sorties HTML conformes WCAG 2.1 AAA [15, 16].

**SPEC.12.02** Chaque élément non textuel (image, graphique, audio, vidéo) doit comporter un texte alternatif ou une transcription.

**SPEC.12.03** Les structures logiques (titres, listes, tableaux) doivent être balisées sémantiquement pour la navigation assistée.

**SPEC.12.04** Les formules mathématiques doivent être exportées en MathML ou dotées d'étiquettes aria-label.

**REQ.13** quickDoc doit offrir une syntaxe visuellement claire et cognitivement homogène.

**SPEC.13.01** Les marqueurs de mise en forme doivent être visibles, non ambigus et uniformes dans tout le langage.

**SPEC.13.02** Une seule notation par fonction doit exister pour éviter les variations dialectales.

**SPEC.13.03** La lisibilité en texte brut doit être conservée ; tout document quickDoc doit rester compréhensible sans rendu [8, 38].

**REQ.14** quickDoc doit être simple à apprendre et à utiliser.

**SPEC.14.01** Un utilisateur novice doit pouvoir rédiger un document basique en moins d'une journée.

**SPEC.14.02** La documentation doit être claire, complète et illustrée d'exemples.

**SPEC.14.03** Chaque ajout syntaxique doit se justifier par un gain explicite de lisibilité ou de puissance expressive [26, 27].

**REQ.15** quickDoc doit favoriser la collaboration et la relecture.

**SPEC.15.01** Les documents doivent pouvoir être versionnés et comparés ligne à ligne dans un VCS (Git).

**SPEC.15.02** Des marqueurs de commentaires ou suggestions doivent être prévus pour l'annotation collaborative.

**SPEC.15.03** Le format texte clair doit permettre la revue sans outil spécifique [26, 40].

## 4.4 Portabilité, performance et sécurité

**REQ.16** quickDoc doit produire plusieurs formats de sortie standardisés.

**SPEC.16.01** Générer au minimum : PDF/A-3, PDF/UA, HTML5 + ARIA, EPUB 3, ODT et LaTeX.

**SPEC.16.02** Le contenu exporté doit conserver la structure logique et les métadonnées.

**SPEC.16.03** Les conversions doivent être contrôlées via une couche d'abstraction (AST → backend).

**SPEC.16.04** Chaque backend doit être testable par une suite de conformité [4, 5].

**REQ.17** quickDoc doit être interopérable et facilement intégrable.

**SPEC.17.01** Fournir un compilateur open-source (licence CeCILL-C).

**SPEC.17.02** Publier une Application Programming Interface (API) publique permettant d'utiliser le parseur et de manipuler l'AST.

**SPEC.17.03** Mettre à disposition une suite de tests de conformité Behavior-Driven Development (BDD) (Cucumber) et un TCK.

**SPEC.17.04** Prévoir des convertisseurs bidirectionnels (Markdown ↔ quickDoc ↔ HTML) [23, 25].

**REQ.18** quickDoc doit présenter de hautes performances sur de grands documents.

**SPEC.18.01** La compilation doit être incrémentale, multi-thread et à consommation mémoire contrôlée.

**SPEC.18.02** Un document de 100 pages avec figures doit se compiler en moins de deux secondes sur une machine standard.

**SPEC.18.03** Les tests de charge doivent inclure des centaines de pages et plusieurs

dizaines de mégaoctets d'images [10, 12].

**REQ.19** quickDoc doit garantir la sécurité lors de l'exécution de code ou du chargement de ressources.

- **SPEC.19.01** Par défaut, aucun code inclus ne doit être exécuté.
- **SPEC.19.02** Le mode d'exécution doit être explicitement activé par l'utilisateur.
- **SPEC.19.03** Les liens externes ne doivent pas être chargés automatiquement sans confirmation.
- **SPEC.19.04** Les configurations d'exception doivent être enregistrées dans un manifeste de sécurité.

**REQ.20** quickDoc doit assurer la compatibilité linguistique et typographique universelle.

- **SPEC.20.01** Support complet d'Unicode, y compris alphabets non latins, symboles scientifiques et émojis.
- **SPEC.20.02** Gestion multilingue et typographie locale (espacement, guillemets, ponctuation).
- **SPEC.20.03** Les citations et références doivent respecter la locale sélectionnée.

## 4.5 Gouvernance, extensibilité et pérennité

**REQ.21** quickDoc doit être gouverné selon un modèle ouvert et transparent.

- **SPEC.21.01** La spécification, les tests et les schémas doivent être publiés sous licence libre.
- **SPEC.21.02** Les évolutions doivent être débattues publiquement et validées par version normative.
- **SPEC.21.03** Les changements syntaxiques majeurs doivent faire l'objet d'une période de transition documentée [23].

**REQ.22** quickDoc doit offrir un système de plugins standardisé pour étendre ses fonctionnalités.

- **SPEC.22.01** Les extensions (ex. nouveaux langages de coloration, exports, filtres) doivent pouvoir être ajoutées sans modifier le cœur du compilateur.
- **SPEC.22.02** Un registre officiel de plugins validés doit être maintenu pour garantir l'interopérabilité.
- **SPEC.22.03** Les API d'extension doivent être stables et documentées [23, 25].

**REQ.23** quickDoc doit assurer la pérennité des documents produits.

- **SPEC.23.01** Les documents doivent être archivables selon les formats ouverts (PDF/A, HTML, Markdown).
- **SPEC.23.02** Les dépendances du langage (polices, modules) doivent être explicitement listées pour garantir la reconstruction future.
- **SPEC.23.03** Un outil de validation doit pouvoir vérifier la conformité syntaxique et sémantique d'un document ancien avec une version donnée du langage.

# 5 Syntaxe

Cette section définit la syntaxe du langage LML avec une grammaire formelle. La grammaire est présentée en notation EBNF simplifiée. Elle décrit la structure d'un document LML en termes de blocs et d'éléments inline, ainsi que la syntaxe des métadonnées et extensions.

## 5.1 Bloc

Un bloc est une structure de haut niveau occupant généralement plusieurs lignes (ex. un paragraphe, une liste, une citation, un bloc de code, une formule affichée, etc.). Un bloc est constituée d'une ligne continue et de la première ligne vide immédiatement en dessous. Un retour chario n'entraine alors pas de création d'un nouveau bloc.

Par défaut, tous les contenus d'un documents constituent des blocs odonnés dans celui-ci.

## 5.2 Balise

Une balise est un élément textuel codifié exprimant un comportement attendu du document. Les balises sont utilisées pour déclarer les mises en formes du texte. Elles peuvent être utilisés :

**Inword** inséré au sein d'un mot et autour d'une ou plusieurs lettres. (ex. première lettre du mot en gras, faute identifiée par un soulignage.)

**Inline** inséré au sein d'un bloc et autout d'un mot ou d'un groupe de mots. (ex. un mot en gras au milieu d'un paragraphe, un symbole mathématique dans une phrase.)

**Inbloc** déclarée en début de ligne, après un retour chario, elle met en forme toute la ligne. (ex. toute une ligne en italique, toute une ligne de formule mathématique.)

**Fullbloc** déclarée en début de bloc et en fin de bloc (ouverture et fermeture du formatage), elle permettent d'appliquer le formatage à l'ensemble du bloc. (ex. un bloc de citation, un bloc de mathématique, un bloc de code).

## 5.3 Métadonnées

Les métadonnées sont des informations structurées concernant le document lui-même ou ses parties (auteur, date, propriétés personnalisées…). Elles peuvent être attachées à n'importe quel bloc. Si elles sont rédigées dans le premier bloc du document, elles sont attachées au document lui-même.

## 5.4 Structure générale d'un document

Un document quickDoc est constitué éventuellement d'un en-tête de métadonnées, suivi d'une suite de blocs de contenu. Le bloc de code 1 présente la représentation EBNF de haut niveau.

```
{
    "Document"    = "(MetadataSection)?", "ContentSection",
    "MetadataSection" = ["MetadataItem"],
    "ContentSection"  = ["Block"]
}
```

**Listing 1 :** *Représentation EBNF au format JSON-LD de la structure d'un document quickDoc*

Ici, la section de métadonnées est optionnelle. Chaque `MetadataItem` est rédigé sous la forme d'une syntaxe `clé:Valeur` et peut être structuré sur plusieurs lignes.

Le corps principal est une suite de `Block`. Les sauts de ligne vides séparent les blocs (similairement à Markdown, un paragraphe est terminé par une ligne vide). Un changement de niveau d'indentation ou des marqueurs spécifiques peuvent également introduire certains blocs (listes, etc., voir ci-après).

## 5.5 Types de contenu

### 5.5.1 Paragraphe et type de texte

Texte libre sur une ou plusieurs lignes. Un paragraphe est le bloc par défaut, aucune syntaxe spéciale n'est nécessaire. Une ligne vide après constitue la fin du paragraphe.

**Symboles courrants** Remplacement à la volée d'associations de charactères par des symboles ou entités courantes en unicode :
— `--` pour un tiret cadratin —,
— `->` flèche droite →,
— `=>` flèche droite à volume ⇒,
— `-----` pour un bloc de séparation monoligne,
— `=====` pour un bloc de séparation doubleligne,

Le texte d'un bloc peut être formaté au niveau d'une lettre, d'un mot, d'une ligne continue ou d'un ensemble de ligne continue espacées entres elles d'un unique retour chariot.

Les éléments inline de LML couvrent les styles de texte, les références internes/externe, les annotations, etc. Ils sont utilisés à l'intérieur des blocs de type paragraphe, titre, légende, cellule de tableau, etc. En voici la liste et la syntaxe :

**Emphase (italique)** délimiteur proposé : / (barre oblique) entourant le texte ciblé. Exemple : `/mot en italique/` produit `<em>mot en italique</em>`.
**Forte emphase (gras)** délimiteur * entourant le texte. Exemple : `*important*` pour `<strong>importa`
**Souligné** délimiteur _ soulignant le texte. Exemple : `_souligné_` pour souligner. (En HTML, cela correspondra à <u> ou style CSS approprié).
**Barré (strike-through)** délimiteur ~ (tilde). Exemple : `\~obsolète\~` donne un texte barré.
**Surligné (highlight)** délimiteur =. Exemple : `=mise en évidence=` pour un texte surligné (couleur de fond différente).
**Exposant (superscript)** délimiteur +. Exemple : `x+2+` transformera le 2 en exposant ($x^2$). Si

on combine plusieurs caractères, par ex +hello+, tout hello sera en exposant. L'exposant d'un exposant est déclaré en imbriquant les délimiteurs mais en utilisant des parenthèses pour séparer les contextes. Exemple : x+(2+3+)+ tranforme le 3 en exposant du 2 et le 2 en exposant du x ($x^{(y^3)}$).

**Indice (subscript)** délimiteur −. servira pour indice.

**Échappement** le caractère \ suivi d'un symbole réservé (comme *, #, etc.) annule sa fonction spéciale et l'insère tel quel. Ex : * dans le texte donnera * au lieu de déclencher du gras.

Les styles ci-dessus peuvent se combiner de manière imbriquée, avec toutefois des restrictions. Par exemple, on peut mettre un texte à la fois en gras et en italique en imbriquant les balises (**texte *mixte***). Certaines combinaisons n'ont pas de sens ou peuvent entrer en conflit. Le tableau 1 définit une matrice de compatibilité des styles imbriqués. Par exemple, combiner exposant et indice sur le même segment produira un middle-script (texte aligné au milieu, usage rare mais valide). Les règles détaillées de combinaison feront l'objet d'un tableau de vérité.

**TABLE 1 :** Compatibility matrix between formatings

| Beacon Type | N° | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Highlight | 1 | x | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| Strike | 2 | 1 | x | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| Subscript | 3 | 1 | 1 | x | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| Superscript | 4 | 1 | 1 | 1 | x | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| Bold | 5 | 1 | 1 | 1 | 1 | x | 1 | 1 | 1 | 0 | 0 | 0 |
| Comment | 6 | 1 | 1 | 1 | 1 | 1 | x | 1 | 1 | 0 | 0 | 0 |
| Italique | 7 | 1 | 1 | 1 | 1 | 1 | 1 | x | 1 | 0 | 0 | 0 |
| Underline | 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | x | 0 | 0 | 0 |
| Verbatim | 9 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | x | 0 | 0 |
| Inline Math | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | 0 |
| Inline Code | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x |

### 5.5.2 Section et Sous-section

Un titre de section commence par un ou plusieurs caractères # suivis d'un espace, à la manière de Markdown. Le nombre de # indique le niveau hiérarchique (1 = Section principal, 2 = sous-section, 3 = sous-sous-section, etc.) limité à 6 niveaux de sections. Un titre peut être suivi en option d'un identifiant unique entre crochets pour les références internes. Cet identifiant pourra être utilisé ailleurs via une référence de type [header:id] pour créer un lien. Exemple :

```
## Méthodologie ;pour un titre de niveau 2;
### Rédaction [id:methodo-redac] ;titre de niveau 3 avec un identifiant;
```

```
# First level header
    ## Second
        ### Third
```

Bien que quickDoc autorise l'indentation des sections jusqu'à 6 niveaux, il convient de maintenir une profondeur de 3 niveaux de titres dans un document pour préserver la fluidité de celui-ci. En règle générale, utiliser un header de niveau 4 implique un besoin de reformulation du contenu pour éviter ce niveau d'indentation.

### 5.5.3 Liste non-ordonnée

Une liste non ordonnée commence par un tiret − suivi d'un espace, à la colonne 0 ou après une indentation correspondant au niveau de nesting (retrait de 4 espaces par niveau). Chaque élément de liste est un bloc pouvant contenir du texte ou d'autres blocs imbriqués. Les styles de puces sont déterminées par les paramètres de rendus (défaut : tiret simple). Exemple de rédaction :

```
- Premier point
    - Sous-point (indenté de 4 espaces)
- Deuxième point

- First list level
^ pas d'espace
    - Second list level
<--> 4 espaces
        - Third list level
<------> 8 espaces
```

A la manière des niveaux de sections, il convient de limiter les listes à 3 niveaux d'indentation. Lorsque le besoin d'ajouter un 4eme niveau d'indentation à une liste non ordonnée se fait resentir, il est probable qu'une représentation graphique sous la forme d'un Work Breakdown Structure (WBS) ou d'une carte mentale serait plus appropriée.

### 5.5.4 Liste ordonnée

Similaire, mais avec un numéro suivi d'un point (ex. `1.`) comme marqueur. La numérotation peut être continuée automatiquement. On peut également utiliser un schéma hiérarchique (ex. `1.1.`) pour les sous-niveaux. Ceci sera formaté en HTML/PDF comme une liste imbriquée. Le style de numérotation (chiffres arabes, lettres, chiffres romains) sera un paramètre de rendu (défaut : chiffres avec point). Exemple de rédaction :

```
1. Premier point
    1.1. Sous-point (indenté de 4 espaces)
2. Deuxième point

1. First ordered list
    1.1. Second
        1.1.1. Third
```

A l'instar des listes non ordonnées, il convient de limiter la constitution de listes ordonnées à 3 niveaux d'indentation. Lorsqu'apparait le besoin d'employer un 4eme niveau, il est probable qu'une représentation graphique, sous la forme d'un diagramme d'activité ou d'un diagramme Business Process Model and Notation (BPMN) soit mieux adapté.

### 5.5.5 Liste de tâches

Une variante de liste à puces où le marqueur peut inclure une case à cocher [ ] (à faire) ou [x] (fait). Ces listes de tâches sont rendues avec des cases cochables en HTML et avec un symbole adéquat en PDF. Exemple de rédaction :

```
- [ ] Écrire l'introduction
- [x] Compiler le document
```

Lists and headers bullets can be followed by a multivalue key in the form of `[<state>:<priority>:<imp` with all optionnal values :

— a formal `<state>` : TODO, PLANED, DOING, PAUSE, DONE, ABORTED,
— an estimated `<priority>` value : A, B, C, D, E,
— an `<impact>` factor : 5, 4, 3, 2 ,1.

```
# [TODO:B:3] Title with a full key

  - [DOING:5] a task with a high impact factor
      - [ ] a check box action
      - [x] a checked check box action
```

### 5.5.6 Annotations et références

**Footnote (note de bas de page)** `[fn:<id>:<text>]` pour insérer un appel de note, avec le contenu de la note défini directement inline.

**Remarque (note de marge)** `[rmq:<id>:<text>]` pour insérer une remarque en marge du document. Le <id> est optionnel mais permet de référencer la remarque et la mentionner dans le corp du document.

**Référence bibliographique** `[cite:<id>,<id>|<display>]` avec `<id>` étant la clé de citation (conforme BibTeX/CSL). Les styles des citations sont configurables au moyen d'une feuille de style CSS. |`<display` est optionnel et peut prendre les valeurs suivantes :
  — `number` : valeur par défaut, la sortis utilisera le mode de citation numéraire (ex. `[12]`.
  — `inline` : la sortie utilisera le mode auteur et date (ex. `[Doe 2020]`).
  — `note` : la sortie utilisera le mode superscript numéraire (ex. $text^{[12]}$).

### 5.5.7 Verbatims et admonition

**Verbatim** délimiteur **:** (deux-points) pour du texte texte littéral à reproduire tel quel (ex. une citation, un exemple). Cela correspondra à un <q> en HTML, rendu monospace.

Un paragraphe pouvant être marqué comme citation (par ex. en commençant par un > comme en Markdown). Ce texte sera rendu en retrait et éventuellement italique. Exemple de rédaction :

```
:: "Le succès est la somme de petits efforts répétés jour après jour."* --
```

corriger l'utilisation de '>' pour alignement avec ':' et intégrer un mécanisme de citation via [cite : @ ...]

types de verbatim : citation (quote), prose (verse), exemple.

```
:::{<parametres>}
 <formule mathématique
 sur plusieurs
 lignes>
:::
```

Les blocs verbatims peuvent être mis en forme en renseignant un type en lieu et place du <parametres> sous la forme `[[[set type:<value>]]]`. Ces types de blocs s'appellent des *admonitions*[24] et peuvent prendre les valeurs suivantes : note, abstract, info, tip, success, question, warning, failure, danger, bug, exemple, quote, keywords, answer, hypothesis, theorem, proof.

Bloc de mise en exergue particulier (note, astuce, avertissement, question, théorème, etc.) généralement rendu avec une icône ou un style distinct.

Il s'agit de blocs spéciaux de note, conseil, avertissement, etc., souvent rendus avec une icône et un style distinctif (fond coloré). Plutôt que d'avoir une syntaxe dédiée pour chaque (comme reST où on écrit .. note: : etc.), LML propose d'utiliser le bloc de verbatim générique avec un paramètre type : indiquant l'admonition. Par exemple :

```
:::{set type:warning}
Ce passage décrit un comportement obsolète.
:::
```

Cet exemple produira un bloc de type "warning/avertissement". La liste des types d'admonitions reconnus inclut au minimum : note, info, tip/astuce, success, question, warning/avertissement, failure/erreur, danger, bug, example/exemple, quote/citation, abstract, ainsi que des types structurés comme theorem, proof, hypothesis, answer pour les documents scientifiques. Chaque type engendrera une mise en forme appropriée dans la sortie (icône, couleur, etc.). L'utilisateur peut étendre/adapter ces styles via la configuration.

### 5.5.8 Code

**Code** délimiteur ` (backtick) entourant un mot ou une phrase de code. Exemple : `` `printf("hello")` `` donnera un <code> en monospace. Pour inclure un backtick littéral dans du code, on pourra utiliser `` `` `` (deux backticks entourant le code si celui-ci contient déjà un backtick). C'est la même règle que Markdown étendu.

Un bloc affichant du code source brut. En quickDoc, on adopte la syntaxe de fences (clôtures) similaires à Markdown : trois accents graves ouvrants déclenchent un bloc de code jusqu'à rencontrer trois accents graves fermants. Après les initiaux, il est possible d'indiquer des paramètres sous la forme `{set key:val ...}` sur la ligne suivante.

```
```{<parametres>}
   <code sur
   plusieurs
   lignes>
```
```

Ceci indique un bloc de code en Python, à exécuter (`play:true`), et à exporter à la fois le code et le résultat (`export:both`). Les paramètres de bloc de code disponibles incluent :

— `lang` (langage pour coloration syntaxique, ex : **python**, **java**, **shell**, etc.),
— `play` (exécution autorisée = true/false),
— `runtime` : moteur ou interpréteur spécifique si besoin, ex : **node** ou **deno** pour JavaScript,
— `export` : choix entre **verbatim**, **result**, **both**, **neither**.

Si `play` n'est pas `true`, le code n'est pas exécuté et seul le code brut est affiché (comme un exemple statique). Le résultat d'un bloc exécuté peut être du texte, une image (graphique), un tableau, etc., qui sera inséré à l'emplacement du bloc.

Les blocs de codes peuvent être configurer de manière à obtenir plusieurs comportements. Les paramètres sont à inscrire sous la forme `[[[set <parameter-name>:<value>]]]` avec les parametres suivants :

— `lang` : défini le langage utilisé pour le formatage du texte,
— `runtime` : défini le moteur d'éxecution le cas échéant et si pertinent (e.g. deno, node, bun, etc.).
— `play` : autorise l'execution du code avec `t` et l'empêche par défaut (`nil`).
— `export` : précise ce qui doit être imprimé lors de l'export. Ce paramètre peut prendre spécifiquement les valeurs suivantes :
  — `verbatim` : imprime le code en police monospace avec son formatage et coloration synthaxique,
  — `result` : remplace le bloc de code avec son résultat (e.g. un graphique, un tableau, etc.),
  — `both` : imprime successivement le code en verbatim puis son résultat,
  — `neither` : n'imprime rien.

```{set lang:python export:result}
    import matplotlib.pyplot as plt
    plt.plot([1, 2, 3], [4, 5, 6])
    plt.show()
```

**Listing 2 :** *Exemple de bloc de code complet*

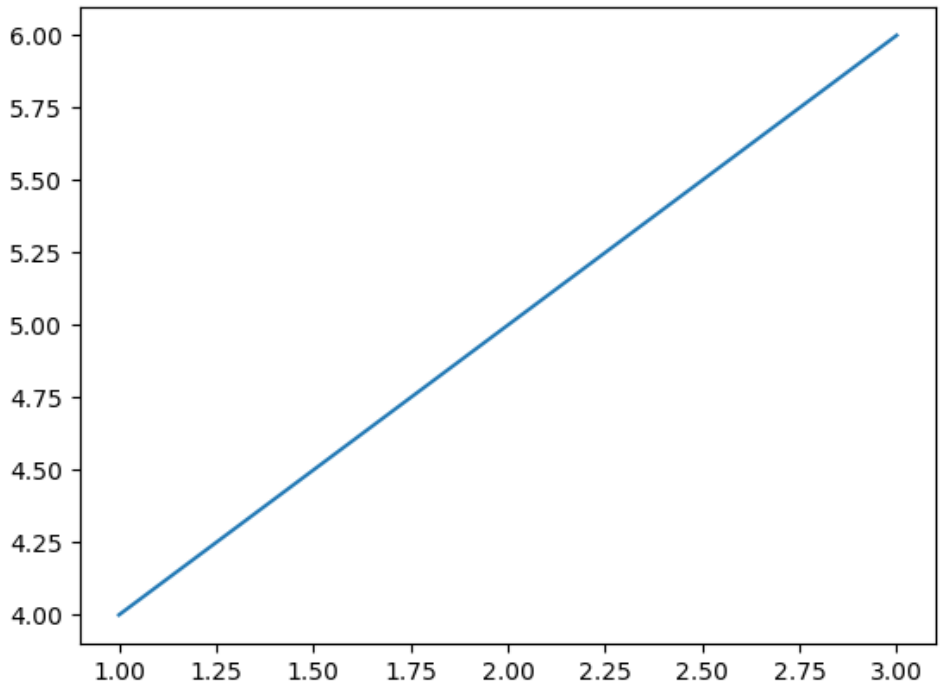


**FIGURE 1 :** Résultat de l'exemple

### 5.5.9 Mathématique

Pour les formules display (isolées au centre). LML utilise également une clôture dédiée, par exemple `$$$` en début et fin de bloc (analogue à `$$ ... $$` en LaTeX, mais sur plusieurs lignes). Entre ces délimiteurs, on place la formule en notation LML (très proche de la syntaxe LaTeX math). Des paramètres peuvent également être fournis après les `$$$` initiaux via `{set ...}`, notamment :

— solver pour spécifier un solveur si on souhaite que la formule soit calculée (ex : solver:sympy),
— éventuellement d'autres paramètres comme le format d'affichage du résultat (exact vs numérique, nombre de décimales, etc. – ces détails pourront évoluer).

Si un solveur est indiqué, la formule sera transmise à ce solveur et son résultat (par ex. simplification ou valeur numérique) pourra être inséré soit en sus, soit à la place, selon le contexte. Par défaut, sans solver, la formule est rendue telle quelle en notation mathématique.

```
$$${<parametres>}
   <formule mathématique
   sur plusieurs
   lignes>
$$$
```

Les blocs de mathématiques peuvent être associées à un résolveur d'équation tel que : `$$ [[[set solver:<nom du résolveur>]]] <equation>` Quelques exemples de résolveurs : Math.js, SymPy, Maxima, SageMath, Wolfram Alpha, etc.

### 5.5.10 Commentaire

**Commentaire** délimiteur `;` (point-virgule) pour mettre un court commentaire non visible dans la sortie, au sein d'une phrase. Ex : `;Note interne;` sera complètement omis à l'export.

**quickDoc** permet des commentaires de l'auteur qui ne seront pas rendus dans la sortie finale. Ceux-ci sont noté par `;;;` en début de bloc et `;;;` en fin. Tout texte à l'intérieur est ignoré lors du rendu. Alternativement, un double point-virgule `;;` en début de ligne peut marquer un commentaire sur la ligne entière (notamment dans un bloc de code, pour commenter du code qui ne sera pas exécuté).

Des commentaires enrichis sont possibles via des paramètres type : sur le bloc de commentaire. Par exemple type:todo ou type:todo-inline pour signaler une tâche à faire. Cela pourra se traduire par un encart "TODO" visible dans le PDF/HTML, éventuellement dans la marge ou dans le flux du texte. Ceci est utile en phase de rédaction (pour indiquer des sections incomplètes, etc.).

```
;;;{<parametres>}
   <commentaires
   sur plusieurs
```

```
   lignes>
;;;
```

Les blocs de commentaires sont de 4 types :

— sans précision, ils ne sont pas exportées
— avec `type:todo-inline` ils impriment un bloc de type ”TODO” en lieu et place ten que celui ci :

bloc ”TODO” en ligne

— avec `type:todo` ils impriment un bloc de type ”TODO” dans la marge du document tel que celui ci :

e.g.

### 5.5.11 Propriétés

Un bloc de propriété permet d’attacher des métadonnées à tout autre bloc.

— inline : `{get/set prop:value}`
— multiline : `{{{get/set prop:value}}}`

Associer des propriétés à un bloc de texte :

```
Ceci est un bloc de texte avec des propriétées attachées.
{{set type:properties key1:value1 key2:value2}}
```

Les propriétés peuvent être utilisés pour collecter des entrées utilisateurs et créer des formulaires en écrivant `{get <property> <key>:<value>}` où `<key>` correspond au type attendu et `<value>` correspond à la contrainte associée. Par exemple, voici des appels d’entrées valides :

— `{get user-name string:20}` : donnera `____________________` et sera affecté à la propriété ”user-name”,
— `{get user-lang string:2}` donnera `__` et sera affecté à la propriété ”user-lang”.

Les déclarations de propriétés monoline sont de la forme `{{get/set prop:value}}`. Elles sont particulièrement adaptées à la déclaration de la mise en page du document (aligmnement du texte, mise en colonne des blocs, etc.). Ces opérations se réalisent en deux étapes :

1. Définir une mise en forme tel que :

   ```
   {{set text_style name:paraghaphe scope:raw-text al:justif size:12 long
   sans-pro columns:nil}}
   ```

2. L’appliquer tel que :

   ```
   {{use text-style name:paragraphe}}
   ```

La mise en forme sera appliquée à partir de sa déclaration (`use`) jusqu’à la déclaration d’un autre style.

### 5.5.12 Tableau

La syntaxe des tableaux peut s'inspirer soit de Markdown (tabulation par des `|`), soit d'AsciiDoc (lignes et colonnes séparées par `|` et formatage avancé via des cellules d'entête `!` etc.). Une ligne d'en-tête optionnelle, encadrée de `|` et séparant les cellules par `|`. Une ligne de séparation en-dessous composée de `|- - -|` (au moins 3 tirets entre chaque `|`) indique la fin de l'en-tête.

Puis les lignes de corps du tableau, avec la même syntaxe de cellule séparées par `|`.

```
| Colonne A | Colonne B |
|-----------+-----------|
| Valeur A1 | Valeur B1 |
| Valeur A2 | Valeur B2 |
```

Les cellules peuvent contenir du texte formaté (italique, etc.) ~~mais pas de blocs multiples (pas de titre ou liste à l'intérieur d'une cellule, sauf en utilisant éventuellement des astuces non couvertes ici).~~ La portée de cette spécification de tableau est ~~limitée aux usages simples de type CSV mis en forme.~~ Les aspects plus complexes (fusion de cellules, tableaux imbriqués) ~~ne sont pas gérés nativement par LML v1, mais pourraient être ajoutés via des extensions~~.

### 5.5.13 Bloc sémantique

utilise les < / >

### 5.5.14 Multimédia

L'insertion d'une image se fera via une directive de lien spécialisée (voir 5.6 sur les liens). Tout lien vers un fichier image (ex : `[[img:chemin/figure.png]]`) insère l'image dans le document. Pour ajouter une légende, on pourra encapsuler cette image dans un bloc de figure, par exemple en la précédant d'un titre de figure ou en utilisant une macro dédiée. Une approche consiste à traiter une image insérée isolée avec un texte de légende suivant immédiatement comme une figure groupée. Par exemple :

```
[[img:diagrams/schéma1.svg]]
{{{set
      id:img1
      title:"Processus illustré"
      alt-text:"Schéma illustratif du processus"
      description:"Une description factuelle de l'image"
}}}
```

### 5.5.15 Snippets

(fragments réutilisables) mécanisme permettant d'injecter du contenu ou des références (ex. inclusion de la valeur d'une propriété).

Tous les blocs de textes peuvent être associées à des tags. Un tag s'écrit `#<tag>` peut être insérée à n'importe quel endroit du bloc de texte en respectant les rêgles de balisage décrites au paragraphe 5.7.

**TABLE 2 :** liste des snippets

| Type | Lemma | Behavior |
|---|---|---|
| Ref Link | `@element` | Insère un `[[lien]]` unidirectionnel |
| Tag | `#tag` | Insère un `[[lien]]` bidirectionnel |

Les tags sont utilisés pour faire de l'analyse sémantique. Ecrire un tag entraine la création ou la mise à jour d'un fichier "tag_<nom-du-tag>.qdo" constitué comme ceci :

```
# Nom-du-tag
[[[get count:?]]] ;; nombre d'occurence dans le projet
[[[get blocs:?nom-du-tag]]] ;; tous les blocs utilisants le tag
```

L'instruction `[[[get blocs:?nom-du-tag]]]` peut être complété par un système de tris. Par exemple, pour lister les blocs par ordre de date décroissante on écrira `[[[get blocs:?nom-du-tag order-by:date-desc]]]`.

Un système de trâme permettant aux utilisateurs de préconfigurer ces documents et de modifier en lot leurs configuration serait un atoût en matière d'expérience utilisateur.

Les snippets sont une fonctionnalité héritée notamment de quickDoc, permettant d'insérer dynamiquement des contenus générés ou de référencer des éléments transverses du document. Ils se présentent sous forme de balises triple-crochets avec mot-clé, par exemple `[[[get ...]]]` ou `[[[set ...]]]`.

### 5.5.1 `[[[set ...]]]` – définition de propriété ou configuration

La balise set est utilisée soit en tête de document pour définir des styles/config globales, soit au sein de blocs (comme vu plus haut) pour paramétrer un bloc spécifique (langage d'un code, type d'admonition, etc.). En général, `[[[set X:Y ...]]]` signifie « assigner la propriété X avec la valeur Y ». Par exemple : `[[[set color:blue]]]`. Dans le cas des blocs, cette instruction apparaît immédiatement après l'ouverture du bloc (comme illustré pour les blocs de code, de maths, etc.). Dans le cas d'une configuration globale, on peut l'utiliser soit dans le front-matter, soit sur une ligne spéciale au début du document éventuellement introduite par un commentaire. QuickDoc montrait une inclusion de fichier de config via `;;[[[set config file:...]]]`, LML pourra avoir plus simplement dans le front-matter une section dédiée pour importer des configs.

En résumé, `[[[set ...]]]` n'est pas exactement un snippet inséré dans le texte final, mais une directive de réglage affectant le rendu ou le comportement.

### 5.5.2 `[[[get ...]]]` – insertion de contenu généré

La balise get permet de récupérer une valeur ou un contenu calculé. On l'utilise au sein du texte pour insérer, par exemple, la valeur d'un compteur, d'une propriété, ou le résultat d'une requête. Syntaxe générale : `[[[get <source> <clé>:<filtre>]]]`.

Exemples envisagés :

`[[[get count:?]]]` pourrait renvoyer un nombre, par ex. le nombre d'éléments correspondant à une requête (voir plus bas).

`[[[get blocs:?tag]]]` pour insérer la liste de tous les blocs taggés par #tag (notion de tag abordée en 5.5.3).

`[[[get property nom]]]` pour insérer la valeur d'une propriété de métadonnée nommée nom définie dans le document (par ex. l'auteur ou le titre).

`[[[get date:now]]]` pour la date du jour, etc., ou d'autres fonctions.

LML devra définir une liste de sources accessibles via get : count (compteurs), blocs (collection de blocs répondant à un critère), property/meta (métadonnées), possiblement env (variables d'environnement ou arguments de compilation), etc.

Des filtres ou paramètres peuvent affiner la requête, p. ex. `[[[get blocs:?tag order-by:date-desc]]]` pour trier les blocs taggés par date décroissante. Ce mécanisme puissant rapproche LML d'un outil de gestion de connaissances, permettant de générer des index, des tables des matières, des listes de tâches automatiques, etc.

5.5.3 Tags et étiquetage sémantique

En LML, on autorise l'ajout de tags à n'importe quel bloc de texte pour une classification sémantique. Un tag s'écrit `#motcle` directement dans le texte ou en préfixe d'un bloc. Par exemple : `#TODO` au début d'une ligne de liste de tâche, ou `#important` dans un paragraphe. Ces tags ne sont pas affichés dans le rendu final, ou éventuellement transformés en éléments visuels discrets, mais surtout ils alimentent une indexation interne.

L'utilisation de tags combinée à `[[[get blocs:?tag]]]` permet de générer par exemple un index de tous les blocs marqués d'un tag particulier. On peut s'en servir pour : liste des TODO restants, index thématique, glossaire (tag `#terme` sur la définition d'un terme), etc. Cette approche en fait un langage plus sémantique et orienté gestion de connaissances.

Techniquement, LML pourrait créer en coulisse des fichiers ou des sections invisibles où sont listés les contenus par tag (comme quickDoc suggère un fichier tag_nom.qdo généré pour chaque tag). La spécification peut rester au niveau conceptuel (il n'est pas nécessaire de normer comment c'est implémenté, juste que le résultat est comme si un tel index existait).

### 5.5.16 Callouts

Les callouts sont des liens bidirectionnels à l'intérieur d'un document et permettent de cibler des blocs. Ils sont utilisés pour sauter rapidement à un contenu, une note, une référence, etc.

### 5.5.17 Liens et références

Les liens sont tous directionnels.

**Référence interne** plusieurs types de callouts permettent de créer des liens internes :

**Table 3 :** Text callouts

| Type | Lemma | Behavior |
|---|---|---|
| Reference | `[ref:id]` | |
| Foot Note | `[fn:id]` | |
| Quote | `[cite:id]` | |
| Figure ref | `[fig:id]` | |
| Table ref | `[tbl:id]` | |
| Code ref | `[src:id]` | |
| Header jump | `[head:id]` | |

- `[ref:ID]` pour référence générique à un élément repéré par l'identifiant ID (section, figure, etc.). Cela affichera soit le numéro de l'élément (ex : "Figure 3") soit un texte par défaut.
- `[fig:ID]` affiche "Figure X" en liant vers l'image de nom ID.
- `[tbl:ID]` pour "Tableau X".
- `[eq:ID]` (éventuellement) pour les équations numérotées "(X)".
- `[src:ID]` pour référencer un listing de code.
- `[header:ID]` pour pointer vers le titre de section identifié.
- `[cite:ID]` insère un renvois vers la bibliographie et référence l'entrée bibliographique.
- `[fn:ID]`
- `[rmq:ID]`

**Liens externes** La syntaxe générale des liens est `[[URL|texte]]` ou `[[URL]]` si pas de texte (affichera l'URL brute ou la ressource intégrée si reconnu). Par exemple : `[[https://example.web]]` pour un lien hypertexte. Si le protocole est img : ou file : ou autre, cela peut déclencher des comportements spécifiques.

Tous les liens suivent l'écriture `[[<CONTEXT>:<LINK>]|[<TEXT>]]` pù seul le `<LINK>` **doit** être renseigné. Les autres éléments sont :

- `<CONTEXT>` fournis des informations complémentaires pour l'affichage du lien. Cela permet de mettre en oeuvre des affichages adaptés aux images, vidéos, player de musique, flux RSS, etc.
- `<LINK>` is the path of the resource on the World Wide Web, in a file system, or on any supported network. We can call a 'Header ID' from the current document or from another one.
- `<TEXT>` est le texte de remplacement à afficher à la place du lien.

un lien peut être attaché à des propriétés `{get}` est utilisé pour renvoyer vers une section particulière du lien (un titre, un callout, etc.). `{set}` est utilisé pour déclarer les éléments de descriptions et d'accessibilité.

**Tag** `#tag`
**Mention** `@someting` renvois à une personne ou à une étape d'un processus
**Radiolink** `[[[name]]]` crée un lien dynamique contextuel renvoyant chaque mentions du `name` du radiolien à la déclaratio nde celui-ci.

**TABLE 4 :** <LINK> types

| Type | Lemma | Can be used to |
|---|---|---|
| URI | | |
| Web URL | `[[https:link]]` | Display a bookmark |
| Local File | `[[file:<path>]]` | |
| Musique | `[[:<path or url>]]` | Display a music player |
| Image | `[[img:<path or url>]]` | |
| Document | `[[doc:<path or url>]]` | Display a document viewer |
| Video | `[[vid:<path or url>]]` | Display a video player |
| Header ID | `[[id:headerId]]` | |
| RSS Flow | `[[rss:<url>]]` | Display a list of last entries |
| IRC Flow | `[[irc:<url>]]` | Display a list of last messages |
| Email | `[[mailto:<email>]]` | Display a contact form |

### 5.6 Normailsation et i18n

insertion des espaces insécables associés au divers éléments au regard des regles lexicales de chaque langues (guillemets, citations, doubles-points, etc.)

### 5.7 Balisage

Lists begins with a bullets that represent their meaning followed by a space. The following is supported :

— Ordered lists starts with `1.`,
— Unordered lists starts with `-`,
— Headers are lists too and starts with a `#`, the number of `#` set the level of the header.

Sublevels (nested lists) are supported for headers, ordered and unordered lists. There must be 4 spaces before the sublevel bullet.

The export backend shall provide settings to customise desired formatting outputs of ordered lists (alpha-numeric numbering, dots, parentheses…)

## 6 Consistency analysis

A consistent lightweight markup language shall have only one way to format text.

Markdown variants on the 5 are limited to those listed by IANA's "Markdown Variant" [21]. We exclude SSW and Quarto, the first one is too contextual and the second is based on Pandoc markdown.

— Bol : Bold
— Ita : Italique
— OrL : Ordered List
— UnL : Unordered List
— Und : Underline
— Hig : Highlight

**TABLE 5 :** Text formatting consistency versus Markdown variants

| Format | Bol | Ita | OrL | UnL | Und | Hig | Str | Ver | Cod | Sup | Sub | Com |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| quickDoc | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| CMD[6] | | | | | | | | | | | | |
| MMD[31] | | | | | | | | | | | | |
| GFM[20] | | | | | | | | | | | | |
| Pandoc[4] | | | | | | | | | | | | |
| Fountain[41] | | | | | | | | | | | | |
| MD for RFCs[42] | | | | | | | | | | | | |
| Pandoc2rfc | | | | | | | | | | | | |
| MDX[43] | | | | | | | | | | | | |
| MyST[34] | | | | | | | | | | | | |
| AsciiDoc | | | | | | | | | | | | |
| reST | | | | | | | | | | | | |
| Org-Mode | | | | | | | | | | | | |
| Textile | | | | | | | | | | | | |
| Djot | | | | | | | | | | | | |
| Wikitext | | | | | | | | | | | | |
| Creole | | | | | | | | | | | | |
| txt2tags | | | | | | | | | | | | |
| Setext | | | | | | | | | | | | |

— Str : Strike
— Ver : Verbatim
— Cod : Inline code
— Sup : Superscript
— Sub : Subscript
— Com : Comment

# 7 Capacity analysis

# 8 Typesystem compatibility

# 9 Conclusion

# 10 Références du document

## 10.1 Liste des figures



## 10.2 Liste des tableaux



## 10.3 Liste des codes sources

# Listings

## 10.4 Liste des glosses

**Backend** Couche logicielle gérant la logique métier, la gestion des données et la communication entre les applications 8, 8, 24

**Interopérabilité** Capacité d'échanger par la présence d'un standard neutre et ouvert des données entre les différents « modèles » sans dépendre d'un acteur ou d'un outil en particulier. 1, 5, 6, 9

## 10.5 Liste des acronymes

**AST** Abstract Syntaxe Tree 4, 7, 8, 8

**API** Application Programming Interface 8, 9

**BPMN** Business Process Model and Notation 14

**BDD** Behavior-Driven Development 8

**LML** Lightweight Markup Language 2, 2, 3, 4, 4, 4, 4, 10, 10, 11, 15, 18, 18, 20, 21, 22, 22, 22, 22

**WBS** Work Breakdown Structure 13

# 11 Bibliographie


[1] Daniel T. Holmes, Mahdi Mobini, and Christopher R. McCudden. “Reproducible Manuscript Preparation with RMarkdown Application to JMSACL and Other Elsevier Journals”. In: *Journal of Mass Spectrometry and Advances in the Clinical Lab* 22 (Nov. 1, 2021), pp. 8–16. DOI: 10.1016/j.jmsacl.2021.09.002. URL: https://www.sciencedirect.com/science/article/pii/S2667145X21000171 (visited on 10/23/2025).

[2] E.F. Haghish. “Markdoc: Literate Programming in Stata”. In: *Stata J.* 16.4 (2016), pp. 964–988. DOI: 10.1177/1536867x1601600409. URL: https://journals.sagepub.com/doi/10.1177/1536867X1601600409 (visited on 10/23/2025).

[3] Lorena A. Barba and Lorena A. Barba. “The Hard Road to Reproducibility”. In: *Science* 354.6308 (Oct. 7, 2016), pp. 142–142. DOI: 10.1126/science.354.6308.142. PMID: 27846503. URL: https://www.science.org/doi/10.1126/science.354.6308.142 (visited on 10/23/2025).

[4] John MacFarlane. *Pandoc User’s Guide*. Standard. Sept. 6, 2025.

[5] Massimiliano Dominici. “An Overview of Pandoc”. In: *TUGboat* 35.1 (2014), pp. 44–50. URL: https://www.tug.org/tugboat/tb35-1/tb109dominici.pdf (visited on 10/23/2025).

[6] John MacFarlane. *CommonMark Spec*. Standard. Version 0.31.2. Jan. 28, 2024. URL: https://spec.commonmark.org/.

[7] Sankalp Khare, Yishan Misra et Venkatesh Choppella. « Using Org-mode and Subversion for Managing and Publishing Content in Computer Science Courses ». In : *2012 IEEE Fourth International Conference on Technology for Education*. 2012 IEEE Fourth International Conference on Technology for Education. Hyderabad, India : IEEE, juill. 2012, p. 220-223. DOI : 10.1109/T4E.2012.58. URL : https://ieeexplore.ieee.org/document/6305975 (visité le 23/10/2025).

[8] Dan Allen and Sarah White. “AsciiDoc Writer’s Guide”. In: ().

[9] David Goodger. *reStructuredText Markup Specification*. Aug. 19, 2025. URL: https://docutils.sourceforge.io/docs/ref/rst/restructuredtext.html (visited on 09/08/2025).

[10] Petr Olšák a Petr Olšák. „TeX in a Nutshell“. In: *Zpravodaj CSTUG* 31.1–4 (1. led. 2021), s. 9–55. DOI: 10.5300/2021-1-4/9. URL: https://dml.cz/handle/10338.dmlcz/150294 (cit. 23. 10. 2025).

[11] Petr Olšák and Petr Olšák. “OpTeX—A New Generation of Plain TeX”. In: *TUGboat* 41.3 (2020), pp. 348–354. DOI: 10.47397/tb/41-3/tb129olsak-optex. URL: https://tug.org/TUGboat/tb41-3/tb129olsak-optex.html (visited on 10/23/2025).

[12] Petr Olšák. “Comparison of OpTeX with Other Formats: LaTeX and ConTeXt”. In: *TUGboat* 42.1 (2021), pp. 44–49. DOI: 10.47397/tb/42-1/tb130olsak-fmtcmp. URL: https://tug.org/TUGboat/tb42-1/tb130olsak-fmtcmp.html (visited on 10/23/2025).

[13] *Ltex-plus/Ltex-Ls-Plus*. LTeX+, 18 sept. 2025. URL: https://github.com/ltex-plus/ltex-ls-plus (visité le 18/09/2025).

[14] *Writefull*. URL: https://www.writefull.com/ (visited on 09/18/2025).

[15] *LaTeX Accessibility Guide*. Engineering Technology Services (ETS). May 30, 2024. URL: https://ets.osu.edu/digital-accessibility/latex-accessibility-guide (visited on 10/16/2025).

[16] Jason C. White. « Using Markup Languages for Accessible Scientific, Technical, and Scholarly Document Creation ». In : *JSESD* 25.1 (15 déc. 2022), p. 1-22. DOI : 10.14448/jsesd.14.0005. URL : https://scholarworks.rit.edu/jsesd/vol25/iss1/5/ (visité le 23/10/2025).

[17] JooYoung Seo and Sean McCurry. “LaTeX Is NOT Easy Creating Accessible Scientific Documents with R Markdown”. In: *Journal on Technology and Persons with Disabilities* 7 (2019), pp. 157–171.

[18] Jens Voegler, Jens Bornschein, and Gerhard Weber. “Markdown – A Simple Syntax for Transcription of Accessible Study Materials”. In: *Computers Helping People with Special Needs*. Ed. by Klaus Miesenberger et al. Vol. 8547. Cham: Springer International Publishing, 2014, pp. 545–548. DOI: 10.1007/978-3-319-08596-8_85. URL: http://link.springer.com/10.1007/978-3-319-08596-8_85 (visited on 10/23/2025).

[19] John Gruber. *Markdown: Syntax*. Daring Fireball. URL: https://daringfireball.net/projects/markdown/syntax (visited on 09/08/2025).

[20] *GitHub Flavored Markdown Spec*. Standard. Version 0.29-gfm. Apr. 6, 2019.

[21] *Markdown Variants*. IANA. Mar. 21, 2023. URL: https://www.iana.org/assignments/markdown-variants/markdown-variants.xhtml (visited on 09/05/2025).

[22] Dan Allen and Sarah White. “AsciiDoc Syntax Quick Reference”. In: ().

[23] *Eclipse Projects / AsciiDoc Language / AsciiDoc Language · GitLab*. Eclipse Projects, 27 août 2025. URL : https://gitlab.eclipse.org/eclipse/asciidoc-lang/asciidoc-lang (visité le 09/09/2025).

[24] Martin Donath. *Admonitions - Material for MkDocs*. URL: https://squidfunk.github.io/mkdocs-material/reference/admonitions/ (visited on 09/09/2025).

[25] *Github/Spec-Kit*. GitHub, 10 sept. 2025. URL : https://github.com/github/spec-kit (visité le 10/09/2025).

[26] Yuyang Liu, Ehsan Noei, and Kelly Lyons. “How ReadMe Files Are Structured in Open Source Java Projects”. In: *Information and Software Technology* 148 (Aug. 1, 2022), p. 106924. DOI: 10.1016/j.infsof.2022.106924. URL: https://www.sciencedirect.

com / science / article / pii / S0950584922000775 (visited on 10/23/2025).

[27] Markus Hofbauer et al. *Large-Scale Collaborative Writing : Technical Challenges and Recommendations*. 17 mars 2023. DOI : 10 . 48550 / arxiv . 2303.09933.

[28] Eagon Meng and Daniel Jackson. *What You See Is What It Does: A Structural Pattern for Legible Software*. 2025. DOI: 10 . 1145 / 3759429 . 3762628. URL: https://arxiv.org/abs/2508.14511 (visited on 10/23/2025). Pre-published.

[29] Martin Junghans et al. "A Grammar for Standardized Wiki Markup". In: *Proceedings of the 4th International Symposium on Wikis*. WikiSym08: 2008 International Symposium on Wikis. Porto Portugal: ACM, Sept. 8, 2008, pp. 1–8. DOI: 10 . 1145 / 1822258 . 1822287. URL: https://dl.acm.org/doi/10.1145/1822258.1822287 (visited on 10/23/2025).

[30] Jens Lechtenbörger. "Emacs-Reveal: A Software Bundle to Create OER Presentations". In: *JOSE* 2.18 (Aug. 4, 2019), p. 50. DOI: 10.21105/jose.00050. URL: https://jose.theoj.org/papers/10.21105/jose.00050 (visited on 10/23/2025).

[31] Fletcher T. Penney. *MultiMarkdown User's Guide*. Standard. Version 6.6.0. May 25, 2023. URL: https://fletcher.github.io/MultiMarkdown-6/.

[32] Héctor Cadavid, Vasilios Andrikopoulos, and Paris Avgeriou. "Improving Hardware/Software Interface Management in Systems of Systems through Documentation as Code". In: *Empir Software Eng* 28.4 (July 6, 2023), p. 100. DOI: 10 . 1007 / s10664 - 023 - 10350 - 7. URL: https://doi.org/10.1007/s10664-023-10350-7 (visited on 10/23/2025).

[33] Héctor Cadavid, Vasilios Andrikopoulos, and Paris Avgeriou. *Documentation-as-Code for Interface Control Document Management in Systems of Systems: A Technical Action Research Study*. June 23, 2022. DOI: 10.48550/arXiv.2206.11668. arXiv: 2206 . 11668 [cs]. URL: http://arxiv.org/abs/2206.11668 (visited on 10/23/2025). Pre-published.

[34] *Jupyter-Book/Mystmd*. Jupyter Book, 9 sept. 2025. URL : https : / / github . com / jupyter - book / mystmd (visité le 09/09/2025).

[35] Ilya V. Schurov. "Qqmbr and Indentml: Extensible Mathematical Publishing for Web and Paper". In: *Proceedings of the 2017 ACM Symposium on Document Engineering*. DocEng '17: ACM Symposium on Document Engineering 2017. DocEng '17. New York, NY, USA: Association for Computing Machinery, Aug. 31, 2017, pp. 121–124. DOI: 10.1145/3103010.3121031. URL: https://dl.acm.org/doi/10.1145/3103010.3121031 (visited on 10/23/2025).

[36] Cezary Kaliszyk and Florian Rabe. *A Survey of Languages for Formalizing Mathematics*. May 26, 2020. DOI: 10.48550/arXiv.2005.12876. arXiv: 2005 . 12876 [cs]. URL: http://arxiv.org/abs/2005.12876 (visited on 10/23/2025). Pre-published.

[37] Ronald Haentjens Dekker et al. “A Schema Language and Parser for Next-Generation Markup Languages”. In: *Balisage Ser. Markup Technol.* Balisage Series on Markup Technologies. Vol. 30. Washington, DC: Mulberry Tecnologies, Inc., 2025. DOI: 10.4242/BalisageVol30.HaentjensDekker01. URL: http://www.balisage.net/Proceedings/vol30/html/HaentjensDekker01/BalisageVol30-HaentjensDekker01.html (visited on 10/23/2025).

[38] Stéphane Ducasse, Laurine Dargaud, and Guillermo Polito. “Microdown: A Clean and Extensible Markup Language to Support Pharo Documentation”. In: International Workshop of Smalltalk Technologies. France, Nov. 2020.

[39] Michael Piotrowski. “A Vision for User-Defined Semantic Markup”. In: *Proc. ACM Symp. Doc. Eng., DocEng*. Proceedings of the ACM Symposium on Document Engineering, DocEng 2019. Berlin Germany: Association for Computing Machinery, Inc, 2019, pp. 1–4. DOI: 10.1145/3342558.3345414. URL: https://dl.acm.org/doi/10.1145/3342558.3345414 (visited on 10/23/2025).

[40] Nicky Bleiel. « Collaborating in GitHub ». In : *2016 IEEE International Professional Communication Conference (IPCC)*. 2016 IEEE International Professional Communication Conference (IPCC). Austin, TX, USA : IEEE, oct. 2016, p. 1-3. DOI : 10.1109/IPCC.2016.7740497. URL : https://ieeexplore.ieee.org/abstract/document/7740497 (visité le 23/10/2025).

[41] *Syntax – Fountain*. URL: https://fountain.io/syntax/ (visited on 09/08/2025).

[42] Thomas Leitner. *Kramdown Syntax*. kramdown. URL: https://kramdown.gettalong.org/syntax.html (visited on 09/08/2025).

[43] *PHP Markdown Extra*. Michel Fortin. URL : https://michelf.ca/projects/php-markdown/extra (visité le 08/09/2025).