EPITA · Projet Rust · 2025-2026

MAL Compressor

MaxAdLe — Format d'archive en Rust

Rapport de Projet — Première Soutenance

Version 1.0.0

Léos Voisin **Adrien Cornille** **Maxime Brague-Héroux**

Mars 2026

1. Répartition des rôles et missions

Le projet MAL Compressor réunit trois membres aux rôles complémentaires, couvrant l'ensemble du cycle de vie du logiciel : du cœur du programme à la distribution finale en passant par la documentation.

Architecture & Cœur Rust

Fichiers : `main.rs`, `lib.rs`, `archive.rs`, `compression.rs`, `utils.rs`

Léos a développé l'intégralité du cœur Rust du projet. Sa mission principale était de concevoir et de mettre en place l'architecture du logiciel, le format d'archive `.mal`, ainsi que toute la logique de compression et de décompression.

Missions réalisées :

— Définition et mise en place du format `.mal` : structure de l'en-tête (`MalHeader`), index des fichiers (`FileEntry`), bloc de données compressées, constantes `MAGIC` et `VERSION`.

— Développement du module de lecture/écriture des données (`archive.rs`) via `serde_json` pour que le format fonctionne de la même façon sur tous les systèmes.

— Mise en place de la compression DEFLATE avec les 10 niveaux (0–9), du calcul CRC32 par fichier avant compression, et de la décompression avec vérification d'intégrité (`compression.rs`).

— Développement des quatre commandes : `compress`, `decompress`, `list` et `verify`.

— Écriture du module utilitaire (`utils.rs`) : collecte des fichiers dans les dossiers via `walkdir`.

— Mise en place du point d'entrée CLI (`main.rs`) avec `clap`, gestion du nommage automatique des archives dans `compressed/` et `decompressed/`, et gestion des doublons par suffixe numérique.

Site web & Distribution

Fichiers : `index.html` (site web), `create_package.sh`

Adrien a développé la vitrine publique du projet et son système de distribution. Sa mission couvrait la communication autour du logiciel ainsi que la création automatique des fichiers à distribuer.

Missions réalisées :

— Conception et développement du site web de présentation du projet en HTML/CSS, servant de point d'entrée pour les téléchargements et la documentation.

— Écriture du script de packaging (`create_package.sh`) : compilation en mode release, détection automatique du système et de l'architecture, création des archives à distribuer (`.tar.gz` et `.zip`) contenant le programme, les scripts, le README et un fichier `INSTALL.txt`.

Scripts, Documentation & Interface

Fichiers (réalisés) : `build.sh`, `compress.sh`, `decompress.sh`, `verify.sh`, `README.md`
À venir : Interface graphique

Maxime a développé les scripts Bash qui permettent d'utiliser le programme sans connaître le terminal, la documentation complète du projet, et travaillera sur une interface graphique pour la deuxième soutenance.

Missions réalisées :

— Écriture du script de compilation (`build.sh`) : vérification de l'installation de Rust, compilation en mode release, copie du programme à la racine du projet.

— Développement des scripts d'aide (`compress.sh`, `decompress.sh`, `verify.sh`) avec gestion des arguments et messages d'erreur clairs.

— Rédaction du `README.md` : guide d'installation pour Linux, macOS et Windows, exemples d'utilisation, description du format `.mal` et de ses algorithmes, FAQ.

Travail prévu pour la deuxième soutenance :

— Développement d'une interface graphique pour rendre le logiciel accessible sans utiliser le terminal.

— L'interface proposera deux boutons principaux — **Compresser** et **Décompresser** — qui ouvriront chacun un explorateur de fichiers permettant de sélectionner des fichiers ou des dossiers à traiter.

2. Planification et état d'avancement

2.1 Organisation du projet

Le projet a été quasiment fini pour la première soutenance. Dès le démarrage, l'équipe a étudié les algorithmes de compression disponibles, défini l'architecture du format `.mal`, puis réparti les tâches selon les rôles décrits en section 1.

Le développement a suivi un ordre logique. Léos a commencé par définir les structures de données et le format `.mal`, ce qui a permis à Maxime d'écrire les scripts Bash en sachant exactement quels arguments le programme attendrait, et à Adrien de créer les archives distribuables une fois le cœur opérationnel. La documentation (`README.md` et site web) a été rédigée en parallèle du développement, en s'appuyant sur le comportement réel du logiciel.

2.2 Ce qui a été réalisé

Toutes les fonctionnalités prévues pour cette première soutenance sont opérationnelles.

Cœur du logiciel (Léos)

Le format d'archive `.mal` est entièrement défini et fonctionnel. Les quatre commandes du programme sont disponibles :

— `compress` : compression de plusieurs fichiers ou dossiers avec choix du niveau (0–9), nommage automatique dans `compressed/` et gestion des doublons.

— `decompress` : extraction dans `decompressed/` avec vérification CRC32 à la décompression.

— `list` : affichage du contenu de l'archive (fichiers, tailles, ratios) en lisant uniquement l'en-tête et l'index, sans décompresser.

— `verify` : vérification de l'intégrité de chaque fichier par comparaison du CRC32 enregistré et recalculé.

Scripts et documentation (Maxime)

Les quatre scripts Bash sont écrits et fonctionnels. Ils permettent une prise en main immédiate sans connaître le CLI Rust. Le `README.md` couvre tous les cas d'utilisation, des exemples concrets et la description technique du projet.

Distribution (Adrien)

Le script `create_package.sh` produit des archives prêtes à être téléchargées, contenant le programme compilé et tous les fichiers nécessaires. Le site web présente le projet avec les liens de téléchargement.

2.3 Ce qui reste à faire

Pour la deuxième et dernière soutenance, Maxime développera une interface graphique permettant d'utiliser le compresseur sans passer par le terminal. Elle proposera deux

boutons principaux — **Compresser** et **Décompresser** — qui ouvriront un explorateur de fichiers pour sélectionner les fichiers ou dossiers à traiter.

3. Choix technologiques et algorithmiques

3.1 Format d'archive `.mal`

Objectif

Définir un format capable de stocker plusieurs fichiers compressés avec leurs informations dans un seul fichier, qui fonctionne de la même façon sur Linux, macOS et Windows.

Choix retenu

> **Format structuré en trois blocs** : en-tête enregistrée en JSON (précédée de sa taille sur 4 octets), index des fichiers, puis blocs de données compressées, chaque bloc étant précédé de sa taille sur 4 octets.

Justification

L'en-tête et l'index sont écrits via `serde_json`, ce qui évite les problèmes de compatibilité entre différentes architectures matérielles. Le format est donc identique et lisible sur tous les systèmes.

L'en-tête contient le *magic number* `MAL\0` pour identifier le format, le numéro de version, et les statistiques globales de l'archive. L'index enregistre pour chaque fichier son chemin, ses tailles avant et après compression, et son CRC32.

Ce découpage *en-tête / index / données* est classique dans les formats d'archives comme ZIP ou TAR, et a un avantage important : la commande `list` ne lit que l'en-tête et l'index sans décompresser les données, ce qui la rend très rapide quelle que soit la taille de l'archive.

3.2 Algorithme de compression : DEFLATE

Objectif

Choisir un algorithme de compression sans perte offrant le meilleur équilibre entre taux de compression, vitesse et facilité d'intégration.

Comparatif des algorithmes

Algorithme	Taux	Vitesse	Complexité	Décision
DEFLATE	Bon (60–80 % texte)	Rapide	Faible	**Retenu**
LZMA (7-Zip)	Excellent	Lent (×10–20)	Élevée	Écarté
Zstandard	Excellent	Très rapide	Très élevée	Écarté
Brotli	Bon (web uniquement)	Lent	Élevée	Écarté
LZ4	Moyen	Extrêmement rapide	Faible	Écarté

Choix retenu

> **DEFLATE** (RFC 1951) via la bibliothèque `flate2`, combinant LZ77 et le codage de Huffman, avec 10 niveaux de compression (0–9).

Justification

DEFLATE combine deux mécanismes. LZ77 repère les données répétées et les remplace par des références dans une fenêtre de 32 KB. Le codage de Huffman attribue ensuite des codes courts aux valeurs fréquentes. Ensemble, ils permettent d'atteindre des taux de compression de 60 à 80 % sur du texte ou du code source, et jusqu'à 95 % sur des données très répétitives.

La bibliothèque `flate2` s'appuie sur zlib, utilisée dans des milliards d'applications : sa fiabilité est bien établie. Les 10 niveaux de compression offrent un bon contrôle du compromis vitesse/taille selon le besoin (niveau 1 pour aller vite, niveau 9 pour la taille minimale).

LZMA aurait donné de meilleurs taux mais est 10 à 20 fois plus lent, ce qui rendrait l'usage quotidien peu agréable. Zstandard est très performant mais plus complexe à intégrer sans avantage décisif ici. DEFLATE est le choix le plus adapté pour ce projet.

3.3 Vérification d'intégrité : CRC32

Objectif

S'assurer qu'un fichier extrait d'une archive est identique au fichier d'origine, et détecter toute corruption accidentelle (transfert réseau, problème de stockage).

Comparatif des méthodes

Méthode	Détection d'erreurs	Vitesse	Décision
CRC32	Très bonne (non cryptographique)	Extrêmement rapide	**Retenu**
MD5	Bonne (collisions connues)	Rapide	Écarté
SHA-256	Excellente (cryptographique)	Lent	Écarté
Adler-32	Correcte	Très rapide	Écarté

Choix retenu

> **CRC32** via la bibliothèque `crc32fast`, calculé sur les données *avant compression*, enregistré dans l'index de l'archive, et vérifié après décompression.

Justification

Le CRC32 est l'algorithme de vérification utilisé par ZIP et PNG. Il détecte 100 % des erreurs simples et doubles, ce qui couvre la grande majorité des cas de corruption rencontrés en pratique.

La bibliothèque `crc32fast` utilise les instructions spécialisées des processeurs modernes pour être très rapide, ce qui rend le calcul du checksum négligeable par rapport au temps de compression.

SHA-256 offrirait des garanties de sécurité inutiles ici : le but est de détecter une corruption accidentelle, pas une modification intentionnelle. MD5 présente des failles connues. CRC32 est donc le bon choix pour ce besoin.

Le checksum est calculé sur les données **avant compression**, ce qui permet de vérifier l'intégrité après décompression sans étape supplémentaire.

3.4 Interface en ligne de commande : `clap`

Objectif

Fournir un programme en ligne de commande structuré en sous-commandes (`compress`, `decompress`, `list`, `verify`), avec validation automatique des arguments et messages d'aide clairs.

Choix retenu

> **clap v4** avec l'API dérivée (`#[derive(Parser)]`) : les sous-commandes et leurs arguments sont définis directement dans le code Rust, ce qui génère automatiquement la gestion des arguments, leur validation et les messages d'aide.

Justification

`clap` est la bibliothèque CLI la plus utilisée dans l'écosystème Rust. L'API dérivée évite tout décalage entre l'aide affichée (`-help`) et le comportement réel du programme : la définition dans le code *est* la documentation. La validation des types (entiers pour le niveau de compression, chemins pour les fichiers) se fait automatiquement avant d'entrer dans la logique du programme, ce qui simplifie le code.

3.5 Scripts Bash et distribution

Objectif

Rendre le logiciel accessible à des utilisateurs sans connaissance du terminal ou de Rust, et automatiser la création des archives à distribuer pour Linux et Windows.

Choix retenu

> Scripts Bash simples encapsulant le programme pour une prise en main immédiate, et script de packaging automatisant la création des archives distribuables.

Justification

Les scripts Bash (`compress.sh`, `decompress.sh`, `verify.sh`) sont la solution la plus simple pour rendre un programme CLI accessible. Ils gèrent la vérification des arguments et affichent des messages d'aide clairs.

Le script `build.sh` vérifie la présence de Rust, compile en mode release et place le programme à la racine — une étape unique à faire une seule fois.

Le script `create_package.sh` détecte automatiquement le système et l'architecture, puis crée une archive contenant le programme compilé, les scripts, le README et un fichier `INSTALL.txt`. L'utilisateur final obtient un outil prêt à l'emploi sans avoir à installer Rust ni à compiler quoi que ce soit.

Conclusion

Le projet MAL Compressor couvre l'ensemble des objectifs fixés pour cette première soutenance. Le cœur Rust de Léos assure les performances et la portabilité du logiciel ; les scripts de Maxime permettent une utilisation simple sans prérequis technique ; le site et le packaging d'Adrien permettent de distribuer le programme facilement.

Les choix retenus — DEFLATE, CRC32, `clap`, `serde_json` — sont tous adaptés à leurs usages et s'appuient sur des bibliothèques fiables et bien établies dans l'écosystème Rust.

Pour la deuxième soutenance, Maxime complétera le projet avec une interface graphique, rendant le compresseur accessible à tous sans passer par le terminal.

4. Références et Bibliographie

Références

[1] Klabnik, S. & Nichols, C. *The Rust Programming Language* — No Starch Press, 2e éd., 2022.
https://doc.rust-lang.org/book/

[2] Crate `flate2` (v1.0) Bindings Rust vers zlib pour DEFLATE.
https://docs.rs/flate2

[3] Crate `clap` (v4) Parseur de ligne de commande pour Rust.
https://docs.rs/clap

[4] Crate `serde` + `bincode` Sérialisation et encodage binaire pour Rust.
https://docs.rs/bincode

[5] Crate `crc32fast` Calcul CRC32 accéléré.
https://docs.rs/crc32fast

[6] Collet, Y. & Kucherawy, M. *Zstandard Compression* — RFC 8878, IETF, 2021.
https://www.rfc-editor.org/rfc/rfc8878

[7] Alakuijala, J. et al. *Brotli Compressed Data Format* — RFC 7932, IETF, 2016.
https://www.rfc-editor.org/rfc/rfc7932

Fin du document — Rapport de soutenance MAL Compressor