Actes de CORIA-TALN-RJCRI-RECITAL 2025
Marseille, France, 30 juin au 4 juillet 2025

Repousser les limites des benchmarks actuels pour une évaluation réaliste des LLMs en migration de code

MALLET Samuel[1, 2] EL KHOURY Joe[2] EGYED-ZSIGMOND Előd[1]
(1) INSA Lyon, 69100 Lyon, France
(2) Onepoint, 29 rue des Sablons, 75116 Paris, France
samuel.mallet@insa-lyon.fr, j.el-khoury@groupeonepoint.com,
elod.egyed-zsigmond@insa-lyon.fr

RÉSUMÉ

Les grands modèles de langage (LLMs) offrent un potentiel important pour la migration de code, mais les benchmarks actuels créent une illusion de maîtrise ne se traduisant pas par de bonnes performances sur des projets industriels complexes. Bien que des avancées comme RepoTransBench incluent des tâches à l'échelle de dépôts complets, ces benchmarks restent irréalistes : taille de projet trop limitée, gestion simplifiée des dépendances, faible diversité technologique et absence de génération ou adaptation automatique des tests. Dans cet article, nous analysons ces limites et nous suggérons de s'inspirer d'approches existantes dans des contextes monolingues, notamment la gestion des contextes longs et la génération automatique de tests, pour concevoir des benchmarks de migration plus réalistes. Notre contribution vise à encourager la communauté à développer des évaluations plus représentatives des défis industriels.

ABSTRACT

Pushing the Boundaries of Current Benchmarks for Realistic LLM Evaluation in Code Migration

Large language models (LLMs) have demonstrated significant potential for code migration tasks. However, current benchmarks fail to represent real-world industrial scenarios, creating an illusion of competence that doesn't translate to actual performance in complex projects. Although recent benchmarks like RepoTransBench offer improvements by including repository-level tasks, they remain unrealistic due to limited project size, simplified dependency management, narrow technological diversity, and absence of test generation or adaptation tasks. We analyze these limitations and suggest leveraging advancements from monolingual contexts—such as managing long-context inputs and automated test generation—to inform future code migration benchmark design. Our contribution encourages the research community to develop more representative benchmarks ensuring that evaluations better reflect LLMs' true capabilities in industrial-scale code migration scenarios.

MOTS-CLÉS : Grand Modèles de Langage, migration de code, évaluation, benchmarks.

KEYWORDS: Large Language Models, code migration, evaluation, benchmarks.

ARTICLE : **Accepté à** la 32e Conférence sur le Traitement Automatique des Langues Naturelles (TALN).

Actes des 32ème Conférence sur le Traitement Automatique des Langues Naturelles (TALN), volume 1 : articles scientifiques originaux, pages 683–697.

1 Introduction

La migration de code vers de nouvelles technologies représente un défi majeur pour les entreprises, motivé par des impératifs techniques (maintenabilité, sécurité, conformité) et stratégiques (obsolescence, pénurie de compétences).

Les grands modèles de langage (LLMs) offrent une approche innovante par rapport aux transpileurs traditionnels (outils convertissant le code d'un langage à un autre dans un paradigme similaire). Leur capacité à traiter simultanément code et langage naturel leur permet de produire un code idiomatique respectant les bonnes pratiques du langage cible, tout en préservant la sémantique et les intentions sous-jacentes dans la documentation, les commentaires et les conventions de nommage.

Pour une adoption industrielle réussie, une évaluation rigoureuse des capacités réelles des LLMs est indispensable afin d'éviter une surestimation de leurs performances et garantir leur adéquation aux exigences concrètes des projets.

Or, en évaluant des tâches simples et isolées, les benchmarks historiques donnent une image faussement optimiste des performances réelles des LLMs. Par exemple, un modèle récent comme GPT-4 atteint des performances très élevées (85,4 %) sur des benchmarks populaires tels que HumanEval (Chen *et al.*, 2021), qui consistent uniquement à implémenter des fonctions individuelles sans interdépendances complexes. Cependant, dès que l'on passe à des tâches plus complexes et représentatives des réalités industrielles, la performance de ce même modèle chute considérablement. Ainsi, sur ClassEval (Du *et al.*, 2023), un benchmark conçu spécifiquement pour évaluer la génération de code au niveau des classes entières, les performances de GPT-4 chutent de 85,4 % à seulement 37 %. Cette dégradation des performances est encore plus prononcée sur le récent benchmark RepoTransBench (Wang *et al.*, 2024), qui se distingue par l'évaluation de la traduction de code au niveau de dépôts entiers, sur lequel GPT-4 obtient une réussite immédiate (Success@1) extrêmement faible de seulement 2,33 % (Wang *et al.*, 2024).

Les défis deviennent encore plus importants lorsqu'il s'agit de code industriel. Comme le souligne l'étude de Ou *et al.* (2024), les performances des LLMs pour la traduction de code ciblant Rust diminue significativement, perdant entre 41,5 % et 56,2 % de leur efficacité initiale lorsqu'ils passent de benchmarks simplifiés à des traductions complètes au niveau du dépôt. Cette étude révèle que les modèles de langage de pointe comme GPT-4 n'atteignent qu'un taux de réussite de 29,3 % pour la traduction de code Python vers Rust dans un contexte de dépôt complet. De même, Nahar *et al.* (2024) identifient plusieurs obstacles majeurs à l'intégration des LLMs dans des produits logiciels industriels, notamment des problèmes liés à l'évaluation des métriques, au manque de méthodes d'évaluation robustes et aux efforts manuels considérables nécessaires pour assurer la qualité.

Un autre défi important concerne la gestion des dépendances et la structure organisationnelle du code. Wang *et al.* (2024) démontrent que les dépôts contenant davantage de fonctions, de classes et d'importations présentent des taux d'échec plus élevés lors de la traduction automatisée. Leur analyse détaillée des erreurs identifie cinq catégories principales de problèmes : les problèmes de fichiers de configuration (44,2 %), la compréhension limitée du code (36,6 %), la génération incomplète (25,3 %), les problèmes liés aux spécificités des langages (14,6 %), et les problèmes d'encodage (8,2 %). Zhang *et al.* (2025) proposent une approche guidée par squelette pour améliorer la traduction de dépôts, mais même avec cette méthode, les meilleurs modèles n'atteignent qu'un taux de réussite de 21 % après plusieurs itérations de débogage, démontrant la complexité persistante de cette tâche.

Dans le contexte industriel, les défis sont également liés aux spécificités des domaines d'application.

Comme le révèle l'enquête d'Urlana *et al.* (2024), plus de 70 % des projets industriels utilisant des LLMs pour la génération de code en sont encore à la phase conceptuelle, montrant le fossé important entre les capacités théoriques et l'application pratique. Leur étude identifie des préoccupations majeures concernant la robustesse (cité par 11 % des praticiens), la sécurité (11 %) et les hallucinations (10 %) comme obstacles principaux à l'adoption industrielle.

Si l'évaluation des LLMs pour la génération de code est désormais un enjeu bien identifié par la communauté scientifique (Chen *et al.*, 2024), les problématiques spécifiques à la migration multilingue de code demeurent sous-explorées. Dans cet article, nous analysons les limites des benchmarks actuels, qui créent une illusion de maîtrise ne se traduisant pas en performances réelles sur des projets industriels. Nos contributions sont : (1) une analyse des benchmarks actuels, révélant leurs insuffisances en termes de taille de projet, gestion des dépendances et diversité technologique, (2) une évaluation des approches émergentes comme celle guidée par squelette, et (3) des pistes d'amélioration inspirées des contextes monolingues, notamment pour la gestion des contextes longs et la génération automatique de tests. Nous encourageons ainsi la communauté à développer des évaluations plus représentatives des défis industriels réels.

2 Évolution des benchmarks : vers une évaluation réaliste des LLMs

Pour comprendre les limitations actuelles des benchmarks de migration de code, il est utile d'examiner l'évolution historique des benchmarks de génération de code, dont les enseignements peuvent inspirer une approche plus réaliste.

Les benchmarks d'évaluation des LLMs pour le code se divisent en deux catégories principales : ceux évaluant la génération à partir de descriptions en langage naturel (NL-to-code), comme HumanEval (Chen *et al.*, 2021) et MBPP (Austin *et al.*, 2021), et ceux évaluant la traduction entre langages de programmation (code-to-code), comme TransCoder-Test (Roziere *et al.*, 2020) et RepoTrans-Bench (Wang *et al.*, 2024). Ces catégories testent des compétences différentes : compréhension des spécifications en langage naturel pour les premiers, et transposition entre syntaxes et paradigmes pour les seconds. Notre analyse couvre ces deux catégories, les enseignements tirés des benchmarks monolingues ayant fortement influencé le développement des benchmarks de migration, enjeu central des applications industrielles.

2.1 Benchmarks monolingues : des fonctions isolées aux scénarios réels

Les benchmarks historiques tels que HumanEval (Chen *et al.*, 2021) et MBPP (Austin *et al.*, 2021) ont longtemps dominé l'évaluation des grands modèles de langage (LLMs) dans la génération de code en Python. Ces benchmarks consistent en tâches relativement simples : générer des fonctions individuelles et isolées à partir d'une spécification en langage naturel. Leur simplicité initiale a permis une adoption massive, mais, comme le soulignent des travaux récents tels que LiveCodeBench (Jain *et al.*, 2024), leur faible complexité est rapidement devenue insuffisante face aux performances croissantes des modèles récents. De plus, la présence potentielle de ces benchmarks statiques dans les vastes corpus d'entraînement des LLMs soulève des préoccupations majeures de contamination des données. Cette contamination risque de fausser l'évaluation réelle des capacités d'un modèle,

un problème que LiveCodeBench (Jain *et al.*, 2024) cherche activement à mitiger en proposant des mises à jour continues des problèmes et des évaluations segmentées dans le temps, basées sur la date de fin d'entraînement des modèles. Certaines études ont d'ailleurs mis en évidence un possible surapprentissage (*overfitting*) des modèles sur des benchmarks comme HumanEval, où les performances ne se généralisent pas toujours à des ensembles de problèmes plus diversifiés ou véritablement nouveaux (Jain *et al.*, 2024).

En réponse à ces limites de saturation et pour viser un plus grand réalisme, des benchmarks évaluant la génération de classes entières avec des interdépendances complexes, tels que ClassEval (Du *et al.*, 2023), ont vu le jour. Plus récemment encore, SWE-Bench (Jimenez *et al.*, 2024) et SWE-Lancer (Miserendino *et al.*, 2025) sont apparus pour évaluer la capacité des modèles à résoudre des problèmes réels de développement logiciel, impliquant des modifications sur des bases de code existantes, plusieurs fichiers et des scénarios industriels concrets. Ces avancées témoignent d'une évolution nette vers un réalisme accru dans les tâches d'évaluation.

Cependant, cette tendance vers le réalisme et la prise en compte des problématiques de contamination, déjà bien amorcée dans les benchmarks monolingues, demeure encore peu mature dans le contexte spécifique de la migration de code entre langages et frameworks différents.

2.2 Benchmarks de migration de code : une évolution parallèle mais tardive

Les premiers benchmarks dédiés à la traduction de code entre langages, tels que TransCoder-Test (Roziere *et al.*, 2020), CodeXGLUE (Lu *et al.*, 2021), ou encore AVATAR (Ahmad *et al.*, 2023), consistaient initialement en tâches simples au niveau des fonctions ou des snippets. Tout comme pour les benchmarks monolingues, leur simplicité a permis d'établir une première référence, mais s'est rapidement avérée insuffisante pour capturer les défis complexes de la migration réelle de code industriel.

Plus récemment, ClassEval-T (Xue *et al.*, 2024) et RepoTransBench (Wang *et al.*, 2024) ont émergé pour évaluer les capacités des LLMs à traduire des classes entières, voire des dépôts complets. ClassEval-T constitue une avancée significative en introduisant une complexité intermédiaire : les modèles doivent gérer des dépendances internes entre méthodes et attributs au sein des classes, ainsi que des dépendances externes avec des bibliothèques tierces. RepoTransBench, quant à lui, pousse encore plus loin le réalisme en testant la traduction à l'échelle de dépôts complets, intégrant la gestion de plusieurs fichiers, de configurations et de structures de projets.

Parmi les approches novatrices récentes, l'approche guidée par squelette (Zhang *et al.*, 2025) mérite une attention particulière. Cette méthode opère en deux étapes : (1) l'extraction et traduction d'un squelette structurel préservant interfaces et dépendances, puis (2) le remplissage guidé du squelette pour la traduction complète. Cette approche offre plusieurs avantages : maintenance incrémentale des traductions, préservation explicite des interfaces, et évaluation à granularité fine malgré des erreurs partielles. Elle introduit notamment une évaluation plus nuancée qui capture des réussites partielles invisibles avec les métriques binaires traditionnelles.

Toutefois, malgré ces progrès, les benchmarks de migration restent nettement en retard par rapport aux approches monolingues. Les limites de ces benchmarks en termes de taille des projets, de gestion réaliste des dépendances, et d'évaluation autonome de génération des tests unitaires restent importantes, comme nous l'analyserons en détail dans la section 4.

Pour une comparaison détaillée des différents benchmarks historiques et récents de migration de code, incluant leur granularité, les langages couverts, les métriques d'évaluation utilisées ainsi que leurs principales limites, nous invitons le lecteur à consulter le tableau comparatif en subsection A.1

3 Méthodologies d'évaluation des LLMs pour la migration de code

L'évaluation des performances des LLMs dans le cadre de la migration de code a connu une évolution parallèle à celle des benchmarks eux-mêmes. Cette section présente les principales approches et métriques utilisées pour juger de l'efficacité des modèles dans ces tâches complexes.

3.1 Évolution des métriques d'évaluation

Les premières approches d'évaluation s'appuyaient principalement sur des métriques de similarité textuelle, héritées du domaine de la traduction automatique naturelle. Des métriques comme BLEU (Papineni *et al.*, 2002) ou CodeBLEU (Ren *et al.*, 2020) comparaient le code généré au code de référence sur la base de similarités n-grammes. Bien que faciles à calculer, ces métriques présentent des limitations fondamentales : deux implémentations syntaxiquement différentes peuvent être fonctionnellement équivalentes, et inversement, un code très similaire à la référence peut contenir des erreurs subtiles mais critiques.

Cette reconnaissance des limites a conduit à l'adoption généralisée d'évaluations basées sur l'exécution. La métrique *pass@k*, introduite avec HumanEval (Chen *et al.*, 2021), évalue si le code généré passe avec succès tous les tests unitaires prédéfinis. Cette approche présente l'avantage considérable de vérifier l'équivalence fonctionnelle plutôt que la similarité syntaxique, et s'est imposée comme standard de facto dans l'évaluation des modèles de génération et de traduction de code (Tao *et al.*, 2024).

3.2 Métriques spécialisées pour la migration de code

Les tâches de migration présentant des défis spécifiques, de nouvelles métriques ont émergé pour évaluer plus précisément la qualité des traductions de code :

- **Computational Accuracy** : Introduite avec TransCoder-Test (Roziere *et al.*, 2020), elle vérifie que la fonction traduite produit les mêmes sorties que la référence pour un ensemble d'entrées de test.
- **DEP (Dependency Evaluation Metrics)** : ClassEval-T (Xue *et al.*, 2024) a introduit des métriques spécifiques (DEP) pour mesurer la capacité des modèles à préserver correctement les relations de dépendance entre les composants d'une classe.
- **Debugging Success Rate@k (DSR@k)** : Proposée par CodeTransOcean (Yan *et al.*, 2023), cette métrique évalue si le code généré peut être corrigé avec succès en k itérations de débogage, reflétant ainsi la facilité avec laquelle les erreurs peuvent être identifiées et corrigées.

Pour l'évaluation au niveau des dépôts complets, des métriques plus complexes ont été introduites. Par exemple, RepoTransBench (Wang *et al.*, 2024) évalue les traductions à l'échelle du dépôt à travers

plusieurs dimensions :

— **Success@k** : Pourcentage de dépôts qui passent tous les tests unitaires dans au moins une des k tentatives, mesurant la capacité du modèle à produire une traduction fonctionnellement équivalente.
— **Build@k** : Pourcentage de dépôts qui compilent avec succès dans au moins une des k tentatives, un prérequis fondamental pour l'exécution.
— **Average Pass Rate (APR)** : Taux moyen de réussite de tous les tests individuels à travers tous les dépôts, offrant une vision plus détaillée que les métriques binaires.

Ces métriques, bien qu'informatives, présentent néanmoins une limitation majeure : leur nature binaire au niveau du dépôt. Un seul échec de compilation dans un composant critique peut empêcher l'exécution de tous les tests, masquant ainsi les parties correctement traduites.

Pour remédier à cette limitation, l'approche guidée par squelette (Zhang *et al.*, 2025) introduit des métriques à granularité plus fine :

— **Build Success Rate** : Proportion de tests unitaires qui compilent avec succès par rapport au nombre total de tests, permettant d'évaluer la qualité de la traduction même lorsque certaines parties du code échouent.
— **Unit Test Success Rate** : Pourcentage de tests unitaires réussis parmi ceux qui compilent, mesurant la correction fonctionnelle des composants traduits.
— **Évaluation modulaire** : Capacité à évaluer indépendamment différents composants du code traduit, permettant d'identifier précisément les réussites et les échecs sans être limité par un échec critique dans une autre partie du code.

Cette dernière approche permet d'identifier précisément les parties correctement traduites même en cas d'échec partiel, offrant ainsi une vision plus nuancée que les métriques binaires traditionnelles — un atout considérable pour l'évaluation de projets complexes et interdépendants. Toutefois, malgré ces avancées en matière de métriques d'évaluation, les benchmarks actuels présentent encore des limitations substantielles qui entravent une évaluation véritablement représentative des scénarios de migration industriels.

4 Limitations des benchmarks actuels et pistes d'exploration futures

Malgré les avancées récentes vers une évaluation plus réaliste des systèmes de migration de code, plusieurs lacunes importantes persistent dans les benchmarks actuels. Ces limitations entravent une mesure fidèle des capacités réelles des LLMs face à la complexité des projets industriels.

4.1 Représentativité insuffisante des scénarios réels

La première limite majeure des benchmarks actuels concerne leur faible représentativité des conditions réelles rencontrées dans les migrations industrielles. Les benchmarks existants, même les plus récents comme RepoTransBench, ne couvrent qu'une fraction restreinte des défis réels auxquels les entreprises sont confrontées. Nous identifions trois dimensions critiques (parmi d'autres) où cette représentativité

fait particulièrement défaut.

A. Taille des bases de code. La taille maximale du contexte d'entrée des grands modèles de langage (LLMs) augmente rapidement, atteignant aujourd'hui plusieurs centaines de milliers, voire millions de tokens (Li *et al.*, 2025). Cette évolution ouvre la voie à des stratégies inédites de migration, où l'intégration d'une base de code complète avec sa documentation dans un seul prompt devient théoriquement possible, offrant potentiellement des gains significatifs par rapport aux méthodes traditionnelles de prétraitement du contexte (Jiang *et al.*, 2024).

Cependant, les benchmarks actuels demeurent inadaptés face aux dimensions réelles des projets industriels. Comme le montre notre analyse détaillée en Annexe C, qui présente le décompte des tokens pour plusieurs projets open-source majeurs, les bases de code industrielles sont considérablement plus volumineuses. Même RepoTransBench, considéré comme l'un des plus réalistes à ce jour, ne représente qu'une fraction infime (moins de 2 %) de la taille moyenne d'un projet industriel, tel que documenté dans notre analyse quantitative en Annexe B qui détaille le nombre de tokens pour chaque dépôt inclus dans ce benchmark. Cette différence d'échelle ne permet pas de tester efficacement les véritables limites des LLMs en contexte industriel. De plus, une étude récente menée avec LongCodeU (Li *et al.*, 2025) montre que les performances réelles des LLMs chutent significativement dès que la longueur du contexte dépasse environ 32 000 tokens, une taille pourtant très inférieure à celle annoncée par les capacités théoriques des modèles récents. Or, 98 des 100 projets contenus dans RepoTransBench restent en dessous de ce seuil (Annexe B), limitant considérablement la pertinence des résultats obtenus sur ce benchmark pour des scénarios industriels réels.

B. Gestion réaliste des dépendances externes. Au-delà de la taille brute du code, la complexité des projets industriels se manifeste également dans leur écosystème de dépendances. Certains benchmarks récents, comme ClassEval-T, incluent partiellement cet aspect en évaluant la capacité des modèles à gérer des bibliothèques externes populaires. Cependant, ces benchmarks présentent une limitation importante : les équivalents dans la technologie cible sont déterminés en amont par les créateurs du benchmark, et non par le modèle lui-même. Or, dans une situation réelle, le système devrait être capable de proposer lui-même ces équivalences, une tâche actuellement non évaluée.

C. Gestion des versions spécifiques de bibliothèques ou frameworks. Étroitement liée à la gestion des dépendances, une troisième dimension essentielle mais quasiment absente des benchmarks actuels est la capacité des LLMs à gérer correctement des versions précises de bibliothèques ou frameworks. Lors d'une migration, une entreprise cible généralement une version précise et stable. Pourtant, aucun benchmark de migration étudié n'évalue cette capacité spécifique. VersiCode (Wu *et al.*, 2024) est l'un des rares à considérer cette dimension, mais son approche, bien que sophistiquée, reste principalement concentrée sur Python.

4.2 Capacités d'évaluation limitées

Les benchmarks existants adoptent généralement une perspective restrictive sur la notion de « solution correcte ». Ils évaluent les modèles sur leur capacité à produire une réponse unique, prédéfinie, ou bien acceptent seulement un nombre très restreint de solutions alternatives. Cette approche ne reflète pas la réalité industrielle, où les migrations impliquent fréquemment des choix multiples, subjectifs ou contextuels. Cette limitation se manifeste principalement à travers deux dimensions critiques.

A. Choix multiples et décisions subjectives. Les migrations réelles impliquent souvent des choix de conception complexes et subjectifs, tels que décider du refactoring de certaines parties du code,

adopter des conventions de nommage précises, sélectionner des patrons de conception particuliers ou réorganiser substantiellement la structure des fichiers. Les benchmarks existants, même ceux évaluant des dépôts complets comme RepoTransBench, imposent généralement une structure prédéfinie avec une solution unique, ou un nombre très limité de solutions acceptables. Cette rigidité ne permet pas d'évaluer la capacité des LLMs à identifier et comparer plusieurs solutions valides, ni à effectuer des choix justifiés en fonction de contraintes spécifiques à chaque projet. Au-delà de cette rigidité structurelle, les benchmarks actuels présentent également des limites importantes dans leur approche de l'évaluation du code généré (discutées plus en détail ci-après et de manière exhaustive en subsection A.1).

B. Génération autonome et adaptation des tests unitaires. Intimement liée à la problématique des choix multiples, la question de la validation du code migré mérite une attention particulière. Actuellement, les benchmarks utilisent généralement des tests unitaires déjà rédigés en amont par les créateurs du benchmark. Cependant, la migration réelle implique souvent de générer ou d'adapter des tests unitaires à partir de la base de code initiale, une tâche complexe nécessitant une compréhension approfondie de la base de code initiale, des différences technologiques, ainsi qu'une capacité de refactoring et de vérification d'équivalence fonctionnelle.

4.3 Obstacles pratiques à la généralisation

Au-delà des problèmes de représentativité et d'évaluation déjà discutés, la généralisation des benchmarks à des contextes variés pose des défis substantiels. Même les benchmarks les plus ambitieux restent limités à un éventail restreint de technologies et langages, entravant leur capacité à représenter fidèlement la diversité technologique industrielle.

A. Diversité technologique insuffisante. La plupart des benchmarks se concentrent sur quelques langages populaires (principalement Python, Java ou JavaScript), mais très peu couvrent une large gamme de frameworks industriels largement utilisés tels qu'AngularJS, React, Vue, Django, ou Rails. Pourtant, pour qu'un outil d'aide à la migration soit véritablement pertinent, il doit être capable d'assister efficacement la migration entre un grand nombre de technologies différentes.

B. Coût humain élevé de construction des benchmarks. Les benchmarks réalistes actuels, tels que SWE-Bench Verified (OpenAI, 2024) (93 ingénieurs mobilisés) ou SWE-Lancer (Miserendino *et al.*, 2025) (100 ingénieurs mobilisés), nécessitent l'intervention coûteuse d'un grand nombre d'ingénieurs expérimentés, limitant fortement leur adaptabilité à d'autres langages ou frameworks. La construction de ClassEval a demandé 500 heures-personnes, tandis que ClassEval-T a nécessité 360 heures-personnes pour la migration manuelle vers Java et C++.

Au-delà de ces défis de diversité technologique et de coût, les approches actuelles d'évaluation peinent encore à capturer plusieurs dimensions cruciales des migrations industrielles, notamment :

- la complexité architecturale inhérente aux systèmes d'entreprise ;
- les contraintes strictes de versions logicielles (LTS, compatibilité) ;
- la nécessité fréquente de moderniser le code au-delà d'une simple transposition fidèle ;
- l'évolution continue des projets durant le processus de migration ;
- et les aspects socio-techniques impactant l'adoption et la maintenance par les équipes.

Ces écarts soulignent le besoin de directions nouvelles pour l'élaboration de benchmarks, que nous aborderons ci-après.

4.4 Pistes d'amélioration et directions futures

Face aux limitations identifiées, plusieurs pistes d'amélioration et directions futures méritent d'être explorées afin de développer des benchmarks de migration de code plus réalistes et pertinents, et ainsi réduire l'écart entre les performances observées en laboratoire et les besoins concrets de l'industrie.

Vers des benchmarks plus représentatifs des défis industriels. Il est crucial de concevoir des benchmarks qui intègrent des bases de code de taille industrielle, reflétant la volumétrie et la complexité des systèmes réels. Cela permettrait d'évaluer rigoureusement les stratégies de gestion des contextes longs et des techniques de récupération d'informations pertinentes. De plus, la gestion des dépendances externes devrait être une composante centrale. L'évaluation porterait non seulement sur la capacité du modèle à identifier et proposer des équivalents de bibliothèques entre différentes technologies, mais aussi, de manière critique, sur son aptitude à générer du code strictement conforme à la version spécifique d'une bibliothèque employée dans le projet cible. Cette dernière compétence, essentielle en contexte industriel où les contraintes de versionnage sont omniprésentes, est actuellement explorée par des initiatives comme VersiCode. Une généralisation de ce type d'évaluation à un plus large spectre de langages et de frameworks, potentiellement enrichie par l'automatisation du scraping des guides de migration et des changelogs comme nous l'avons évoqué, constituerait une avancée significative.

Adopter des méthodologies d'évaluation plus flexibles et complètes. Pour dépasser la rigidité des solutions uniques, l'utilisation de tests d'intégration end-to-end, à l'instar de SWE-Lancer, offrirait une évaluation plus holistique de la fonctionnalité du code migré, tout en laissant au modèle la liberté de prendre des décisions de conception. Parallèlement, la capacité des LLMs à générer ou adapter eux-mêmes les tests unitaires (en s'inspirant par exemple de TDD Bench (**?**)) devrait être une nouvelle dimension d'évaluation, reflétant une étape essentielle du processus de migration réel.

Réduire les coûts de création et favoriser la diversité technologique. Pour surmonter le coût prohibitif de la création manuelle de benchmarks, des approches semi-automatisées, comme celles de NaturalCodeBench (Zhang *et al.*, 2024) combinant génération automatique et validation humaine légère, sont prometteuses. Cela faciliterait l'extension des benchmarks à une plus grande diversité de langages et, surtout, de frameworks industriels (Angular, React, Vue, Django, Spring, etc.).

Intégrer les dimensions industrielles et socio-techniques. Enfin, les futurs benchmarks devraient s'efforcer d'intégrer les contraintes spécifiques aux environnements industriels, telles que la nécessité de cibler des versions LTS, d'assurer la compatibilité descendante, ou de moderniser le code au-delà d'une simple traduction fidèle. Bien que plus difficiles à quantifier, la prise en compte des aspects socio-techniques, comme la facilité de compréhension du code migré par les équipes ou la qualité de la documentation générée, enrichirait également la pertinence des évaluations.

En explorant ces pistes, la communauté scientifique pourra développer des cadres d'évaluation qui non seulement mesurent plus justement les capacités réelles des LLMs en migration de code, mais orientent également leur développement vers des solutions véritablement adaptées aux défis complexes et multidimensionnels de l'industrie.

5 Conclusion

La migration de code demeure un enjeu majeur pour les entreprises, et les LLMs offrent une solution potentiellement transformative à ce défi. Cependant, l'écart préoccupant entre les évaluations académiques et les exigences des migrations industrielles réelles crée une illusion de compétence qui risque de conduire à des échecs coûteux. Les benchmarks actuels constituent des progrès significatifs mais restent insuffisants face à la complexité multidimensionnelle des migrations industrielles. La chute drastique des performances observée lors du passage des fonctions isolées aux dépôts complets révèle l'inadéquation fondamentale des métriques et méthodologies d'évaluation.

Ces limitations sont exacerbées par d'importants obstacles pratiques à la généralisation. La diversité technologique insuffisante des benchmarks actuels, qui se concentrent principalement sur quelques langages populaires, ne permet pas d'évaluer adéquatement les migrations entre les nombreux frameworks industriels (AngularJS, React, Django, Rails, etc.) couramment utilisés en entreprise. De plus, le coût humain élevé de construction de benchmarks réalistes, nécessitant l'intervention de nombreux ingénieurs expérimentés, limite fortement leur adaptabilité à différentes technologies et contextes.

Pour combler cet écart, nous avons listé plusieurs pistes prometteuses : approches hybrides combinant automatisation et validation humaine pour réduire le coût humain, benchmarks stratifiés couvrant une plus grande diversité technologique, métriques contextuelles permettant d'évaluer des choix multiples et subjectifs, et prise en compte des aspects socio-techniques essentiels dans les migrations industrielles. La construction semi-automatisée de cas de test, l'utilisation de tests end-to-end plutôt que strictement unitaires, et l'évaluation autonome de génération des tests représentent également des axes d'amélioration significatifs.

Ces recommandations visent à encourager le développement de méthodologies d'évaluation qui reflètent plus fidèlement la réalité industrielle, permettant ainsi de mieux guider l'évolution future des modèles dans ce domaine crucial et de réduire l'écart entre performances académiques rapportées et applicabilité industrielle réelle.

Références

AHMAD W. U., TUSHAR M. G. R., CHAKRABORTY S. & CHANG K.-W. (2023). AVATAR : A parallel corpus for Java-python program translation. In A. ROGERS, J. BOYD-GRABER & N. OKAZAKI, Éds., *Findings of the Association for Computational Linguistics : ACL 2023*, p. 2268–2281, Toronto, Canada : Association for Computational Linguistics. DOI : 10.18653/v1/2023.findings-acl.143.

ATHIWARATKUN B., GOUDA S. K., WANG Z., LI X., TIAN Y., TAN M., AHMAD W. U., WANG S., SUN Q., SHANG M., GONUGONDLA S. K., DING H., KUMAR V., FULTON N., FARAHANI A., JAIN S., GIAQUINTO R., QIAN H., RAMANATHAN M. K., NALLAPATI R., RAY B., BHATIA P., SENGUPTA S., ROTH D. & XIANG B. (2023). Multi-lingual evaluation of code generation models. *arXiv preprint arXiv :2210.14868*.

AUSTIN J., ODENA A., NYE M., BOSMA M., MICHALEWSKI H., DOHAN D., JIANG E., CAI C., TERRY M., LE Q. *et al.* (2021). Program synthesis with large language models. *arXiv preprint arXiv :2108.07732*.

CABALLERO E., OPENAI . & SUTSKEVER I. (2016). Description2Code Dataset. DOI : 10.5281/zenodo.5665051.

CASSANO F., GOUWAR J., NGUYEN D., NGUYEN S., PHIPPS-COSTIN L., PINCKNEY D., YEE M.-H., ZI Y., ANDERSON C. J., FELDMAN M. Q., GUHA A., GREENBERG M. & JANGDA A. (2022). Multipl-e : A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv :2208.08227*.

CHEN L., GUO Q., JIA H., ZENG Z., WANG X., XU Y., WU J., WANG Y., GAO Q., WANG J. *et al.* (2024). A survey on evaluating large language models in code generation tasks. *arXiv preprint arXiv :2408.16498*.

CHEN M., TWOREK J., JUN H., YUAN Q., PINTO H. P. D. O., KAPLAN J., EDWARDS H., BURDA Y., JOSEPH N., BROCKMAN G. *et al.* (2021). Evaluating large language models trained on code. *arXiv preprint arXiv :2107.03374*.

DU X., LIU M., WANG K., WANG H., LIU J., CHEN Y., FENG J., SHA C., PENG X. & LOU Y. (2023). Classeval : A manually-crafted benchmark for evaluating llms on class-level code generation.

JAIN N., HAN K., GU A., LI W.-D., YAN F., ZHANG T., WANG S., SOLAR-LEZAMA A., SEN K. & STOICA I. (2024). Livecodebench : Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv :2403.07974*.

JIANG J., WANG F., SHEN J., KIM S. & KIM S. (2024). A survey on large language models for code generation. *arXiv preprint arXiv :2406.00515*.

JIAO M., YU T., LI X., QIU G., GU X. & SHEN B. (2023). On the evaluation of neural code translation : Taxonomy and benchmark. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, p. 1529–1541 : IEEE.

JIMENEZ C. E., YANG J., WETTIG A., YAO S., PEI K., PRESS O. & NARASIMHAN K. R. (2024). SWE-bench : Can language models resolve real-world github issues ? In *The Twelfth International Conference on Learning Representations*.

KHAN M. A. M., BARI M. S., DO X. L., WANG W., PARVEZ M. R. & JOTY S. (2024). XCodeEval : An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In L.-W. KU, A. MARTINS & V. SRIKUMAR, Éds., *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1 : Long Papers)*, p. 6766–6805, Bangkok, Thailand : Association for Computational Linguistics. DOI : 10.18653/v1/2024.acl-long.367.

LI J., GUO X., LI L., ZHANG K., LI G., TAO Z., LIU F., TAO C., ZHU Y. & JIN Z. (2025). Longcodeu : Benchmarking long-context language models on long code understanding. *arXiv preprint arXiv :2503.04359*.

LU S., GUO D., REN S., HUANG J., SVYATKOVSKIY A., BLANCO A., CLEMENT C., DRAIN D., JIANG D., TANG D., LI G., ZHOU L., SHOU L., ZHOU L., TUFANO M., GONG M., ZHOU M., DUAN N., SUNDARESAN N., DENG S. K., FU S. & LIU S. (2021). CodeXGLUE : A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

MISERENDINO S., WANG M., PATWARDHAN T. & HEIDECKE J. (2025). Swe-lancer : Can frontier llms earn $1 million from real-world freelance software engineering ? *arXiv preprint arXiv :2502.12115*.

NAHAR N., KÄSTNER C., BUTLER J., PARNIN C., ZIMMERMANN T. & BIRD C. (2024). Beyond the comfort zone : Emerging solutions to overcome challenges in integrating llms into software products.

OpenAI (2023). tiktoken : A fast bpe tokenization library. Version 0.9.0.

OpenAI (2024). Présentation de swe-bench verified. Consulté le 11 mars 2025.

Ou G., Liu M., Chen Y., Peng X. & Zheng Z. (2024). Repository-level code translation benchmark targeting rust.

Papineni K., Roukos S., Ward T. & Zhu W.-J. (2002). Bleu : a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, p. 311–318, USA : Association for Computational Linguistics. doi : 10.3115/1073083.1073135.

Puri R., Kung D. S., Janssen G., Zhang W., Domeniconi G., Zolotov V., Dolby J., Chen J., Choudhury M., Decker L., Thost V., Buratti L., Pujar S., Ramji S., Finkler U., Malaika S. & Reiss F. (2021). Codenet : A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv :2105.12655*.

Ren S., Guo D., Lu S., Zhou L., Liu S., Tang D., Sundaresan N., Zhou M., Blanco A. & Ma S. (2020). Codebleu : a method for automatic evaluation of code synthesis.

Rithy I. J., Hossain Shakil H., Mondal N., Sultana F. & Shah F. M. (2022). Xtest : A parallel multilingual corpus with test cases for code translation and its evaluation*. In *2022 25th International Conference on Computer and Information Technology (ICCIT)*, p. 623–628. doi : 10.1109/ICCIT57492.2022.10055851.

Roziere B., Lachaux M.-A., Chanussot L. & Lample G. (2020). Unsupervised translation of programming languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA : Curran Associates Inc.

Tao Q., Yu T., Gu X. & Shen B. (2024). Unraveling the potential of large language models in code translation : How far are we ? *arXiv preprint arXiv :2410.09812*.

Urlana A., Kumar C. V., Singh A. K., Garlapati B. M., Chalamala S. R. & Mishra R. (2024). Llms with industrial lens : Deciphering the challenges and prospects–a survey. *arXiv preprint arXiv :2402.14558*.

Wang Y., Wang Y., Wang S., Guo D., Chen J., Grundy J., Liu X., Ma Y., Mao M., Zhang H. & Zheng Z. (2024). Repotransbench : A real-world benchmark for repository-level code translation. *arXiv preprint arXiv :2412.17744*.

Wu T., Wu W., Wang X., Xu K., Ma S., Jiang B., Yang P., Xing Z., Li Y.-F. & Haffari G. (2024). Versicode : Towards version-controllable code generation.

Xue P., Wu L., Yang Z., Wang C., Li X., Zhang Y., Li J., Jin R., Pei Y., Shen Z., Lyu X. & Keung J. W. (2024). Escalating LLM-based Code Translation Benchmarking into the Class-level Era. *arXiv e-prints*, p. arXiv :2411.06145. doi : 10.48550/arXiv.2411.06145.

Yan W., Tian Y., Li Y., Chen Q. & Wang W. (2023). CodeTransOcean : A comprehensive multilingual benchmark for code translation. In H. Bouamor, J. Pino & K. Bali, Éds., *Findings of the Association for Computational Linguistics : EMNLP 2023*, p. 5067–5089, Singapore : Association for Computational Linguistics. doi : 10.18653/v1/2023.findings-emnlp.337.

Zhang S., Zhao H., Liu X., Zheng Q., Qi Z., Gu X., Dong Y. & Tang J. (2024). NaturalCodeBench : Examining coding performance mismatch on HumanEval and natural user queries. In L.-W. Ku, A. Martins & V. Srikumar, Éds., *Findings of the Association for Computational Linguistics : ACL 2024*, p. 7907–7928, Bangkok, Thailand : Association for Computational Linguistics. doi : 10.18653/v1/2024.findings-acl.471.

ZHANG X., WEN J., YANG F., ZHAO P., KANG Y., WANG J., WANG M., HUANG Y., NALLIPOGU E., LIN Q. *et al.* (2025). Skeleton-guided-translation : A benchmarking framework for code repository translation with fine-grained quality evaluation. *arXiv preprint arXiv :2501.16050*.

ZHENG Q., XIA X., ZOU X., DONG Y., WANG S., XUE Y., SHEN L., WANG Z., WANG A., LI Y. *et al.* (2023). Codegeex : A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, p. 5673–5684.

ZHU M., JAIN A., SURESH K., RAVINDRAN R., TIPIRNENI S. & REDDY C. K. (2022a). Xlcost : A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv :2206.08474*.

ZHU M., SURESH K. & REDDY C. K. (2022b). Multilingual code snippets training for program translation. *Proceedings of the AAAI Conference on Artificial Intelligence*, **36**(10), 11783–11790. DOI : 10.1609/aaai.v36i10.21434.

A Annexes

A.1 Tableau comparatif

Table 1: Comparaison des benchmarks d'évaluation des LLMs pour la génération et migration de code

Nom du benchmark	Granularité	Langages couverts	Métrique(s)	Source méthode(s) d'évaluation	Origine des données	Limitations principales
TransCoder-Test (2020) (Roziere *et al.*, 2020)	Fonctions statiques	C++, Java, Python	Tests E/S	Entrées générées semi-aléatoirement	GeeksForGeeks	Problèmes simples, fonctions isolées, pas de dépendances
CodeTrans (CodeXGLUE) (2021) (Lu *et al.*, 2021)	Méthode	Java ↔ C#	BLEU, CodeBleu, EM	Variable selon la tâche	Projets open source traduits	Métrique de similarité textuelle, pas d'exécution
CodeNet (2021) (Puri *et al.*, 2021)	Fonction	55 langages. Mais 95 % des données en C++, Python, Java, C, Ruby, et C#	Tests E/S	E/S collectées depuis les sources	AIZU et AtCoder	Problèmes algorithmiques, Bruit, 50 % incorrects selon experts (Zhu *et al.*, 2022a)
AVATAR (2021) (Ahmad *et al.*, 2023)	Fonction & Programme	Java ↔ Python	Tests E/S pour une fraction du dataset + Similarité syntaxique	E/S de AtCoder et TransCoder-Test	Sites de compétitions, plateformes en ligne, GitHub	Problèmes algorithmiques, contexte simplifié
CoST (2022) (Zhu *et al.*, 2022b)	Snippet & Programme	C++, Java, Python, C#, JS, PHP, C	BLEU, CodeBLEU	GeeksForGeeks	GeeksForGeeks	Pas d'évaluation fonctionnelle, snippets isolés
XLCoST (2022) (Zhu *et al.*, 2022a)	Snippet & Programme	C++, Java, Python, C#, JS, PHP, C	BLEU, CodeBLEU	GeeksForGeeks	GeeksForGeeks	Problèmes simples, pas de vérification fonctionnelle
MultiPL-E (2022) (Cassano *et al.*, 2022)	Fonction	18+ langages	Tests E/S	E/S de HumanEval et MBPP	HumanEval/MBPP traduits	Fonctions isolées, traduction manuelle de tests
MBXP & Multilingual HumanEval (2022) (Athiwaratkun *et al.*, 2023)	Fonction	13 langages	Tests E/S	Traduction automatique des tests de MBXP et HumanEval	Transpilation des tests de MBPP et HumanEval	Fonctions isolées, problèmes basiques
XTest (2023) (Rithy *et al.*, 2022)	Programme	C, C++, C#, Java, Python, PHP, Ruby, Go, JavaScript	BLEU/CodeBLEU et 20 % couvert par test unitaires	Tests unitaires collectés depuis Description2Code (Caballero *et al.*, 2016)	Sites de programmation compétitive	Problèmes compétitifs seulement, Non open-source
xCodeEval (2023) (Khan *et al.*, 2024)	Document-level	C, C#, C++, Go, Java, JavaScript, Kotlin, PHP, Python, Ruby, Rust	Tests unitaires	CodeForces	CodeForces	Problèmes algorithmiques uniquement, source unique de données
HumanEval-X (2023) (Zheng *et al.*, 2023)	Fonction	Python, C++, Java, JS, Go	Tests Unitaires	Réécriture manuelle des tests de HumanEval	Traduction de HumanEval	Fonctions isolées, petit dataset (164 problèmes)
G-TransEval (2023) (Jiao *et al.*, 2023)	Fonction	Python, C++, Java, C#, JavaScript	BLEU, CodeBLEU, Tests E/S	Ecriture manuelle des tests	GeeksForGeeks, HumanEval, extraits .NET	Problèmes algorithmiques
MultilingualTrans (2023) (Yan *et al.*, 2023)	Programme	Python, C, C++, C#, Java, Go, PHP, Visual Basic	Exact Match (EM), BLEU, CodeBLEU	Rosetta Code	Rosetta Code	Pas de vérification fonctionnelle automatique, métriques basées sur la similarité uniquement
NicheTrans (2023) (Yan *et al.*, 2023)	Programme	37 langages niche vers 8 populaires	Exact Match (EM), BLEU, CodeBLEU	Rosetta Code	Rosetta Code	Pas de vérification fonctionnelle automatique, métriques basées sur la similarité uniquement
LLMTrans (2023) (Yan *et al.*, 2023)	Programme	C, C++, C#, Visual Basic, Go, PHP, Java vers Python	Debugging Success Rate@K (DSR@K)	Tests E/S	Rosetta Code	Très petit (350 échantillons), traduction uniquement vers Python
PolyHumanEval (2024) (Tao *et al.*, 2024)	Fonction	14 langages	Tests E/S	Adaptation automatique des tests de HumanEval	HumanEval	Fonctions isolées, pas de dépendances externes
ClassEval-T (2024) (Xue *et al.*, 2024)	Classe	Python, Java, C++	Tests E/S (CA), Tests de Compilation (CSR)	Traduction des tests de ClassEval	ClassEval traduit manuellement	Support pour 3 langages uniquement, petit dataset (94 classes), mapping manuel des dépendances
RustRepoTrans (2024) (Ou *et al.*, 2024)	Dépôt	C/C++, Java, et Python vers Rust	Tests Unitaires	Tests Unitaires existants	Open Source projets GitHub	Tâches focalisées sur des fonctions extraites
RepoTransBench (2024) (Wang *et al.*, 2024)	Dépôt	Java→Python	Tests Unitaires	Traduction de tests semi-automatisée	*The Stack* & *The Stack v2*	Mapping rigide fichier-fichier, Score d'évaluation binaire
TransRepo-BENCH (2025) (Zhang *et al.*, 2025)	Dépôt	Java→C#	Tests unitaires, Tests de Compilation	Traduction semi-automatique des squelettes et tests Java	GitHub (Java-Design-Patterns [6])	Effort manuel significatif pour la correction des squelettes et des tests, actuellement limité à Java vers C#, Depôts Java éducatifs, non réalistes

Note : plusieurs de ces benchmarks proposent des tâches pertinentes mais non liées à la traduction (Code retrieval, code classification...). Nous nous intéressons seulement à la traduction, et les colonnes du tableau ne se concentrent que sur les caractéristiques du benchmark liées à la traduction. De plus, nous nous intéressons seulement aux métriques automatisées.

B Nombre détaillé de tokens par dépôt dans RepoTransBench

Le graphique suivant présente la distribution du nombre total de tokens pour chaque dépôt inclus dans le benchmark RepoTransBench. Ces chiffres correspondent uniquement à la somme des tokens des fichiers Python (code source et tests unitaires), à l'exclusion des fichiers de configuration ou de données. Le calcul des tokens a été effectué avec la bibliothèque `tiktoken` (OpenAI, 2023), en utilisant l'encodage `o200k_base`.

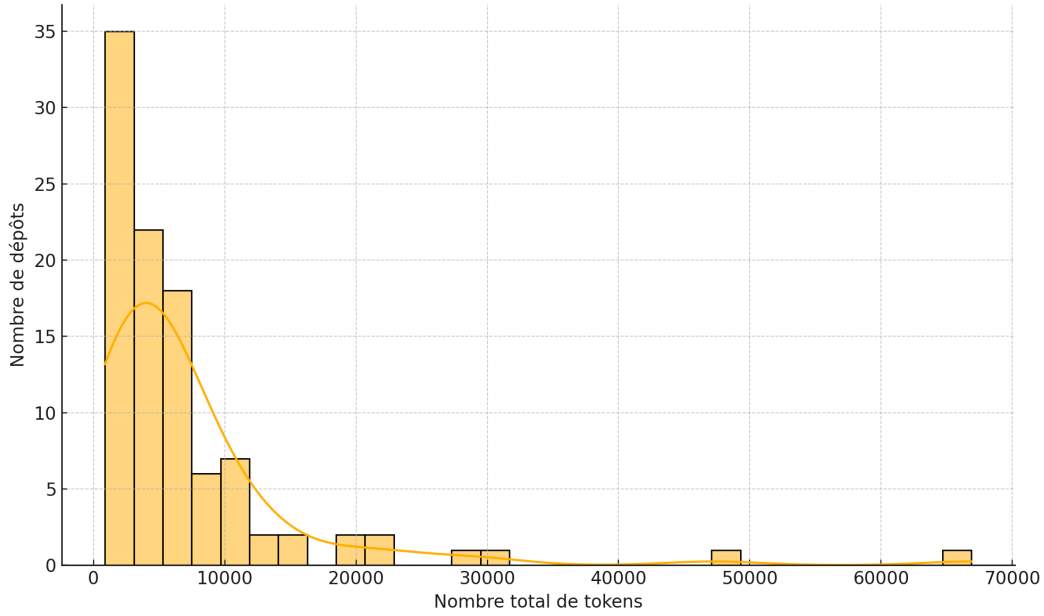

FIGURE 1 – Distribution du nombre total de tokens par dépôt dans RepoTransBench.

C Exemples de tailles de bases de code industrielles

Les benchmarks actuels de migration de code utilisent principalement des bases de code de taille limitée, bien en deçà des projets industriels réels. Afin d'illustrer cet écart, nous listons ci-dessous quelques projets open-source populaires avec un calcul du nombre total de tokens pour les types de fichiers principaux. Ces mesures ont été calculées le **20 mars 2025** en utilisant la librairie `tiktoken` et l'encodage `o200k_base`.

Référentiel	Fichiers considérés	Nombre total de tokens
Chart.js	ts, js, html	510 986
TensorFlow	cpp, py	11 347 669
Linux Kernel	c	228 983 249
Django	py	3 908 517
Pydantic	py	900 597
Expensify	ts	2 153 878

TABLE 2 – Estimation du nombre de tokens dans plusieurs projets open-source de grande envergure