Cigref
RÉUSSIR
LE NUMÉRIQUE

CAST

Do Tank « Impacts de l'IA sur les activités d'ingénierie logicielle »

GT 3/6
« Modernisation des applications »

Janvier 2026

1. Introduction
2. Opportunités de l’IA pour la modernisation des applications
3. Conclusion

Les sociétés ayant participé au GT

Pilote du GT :
Thierry Fraudet

01

Introduction

Modernisation des applications : de quoi parle-t-on ?

La modernisation d'une application englobe de nombreux cas d'usage, depuis la simple correction de code patrimonial (*legacy*) jusqu'à la réécriture complète d'une application dans une nouvelle architecture cloud-native. Pour structurer cette diversité, nous proposons le découpage suivant en cinq niveaux de complexité croissante :

Complexité

I – Amélioration ponctuelle de la qualité	II – Amélioration de la sécurité, conformité, et mise à niveau des composants	III – Modernisation des frameworks et des langages standards	IV – Modernisation des frameworks « maison »	V – Réécriture
Corrections ciblées du code existant, et refactorisation limitée sans modification de l'architecture ou des technologies utilisées. Impact localisé sur des modules ou fonctionnalités spécifiques.* *(*) processus de modification du code qui permet de faciliter sa maintenance, sa compréhension et son extension, sans modifier son comportement.*	*Mise à jour des bibliothèques, dépendances et composants tiers pour corriger les vulnérabilités. Application de correctifs de sécurité et montée de version des frameworks en place.*	*Migration vers des versions récentes des frameworks et langages actuels. Évolution significative de la stack technique (désigne l'ensemble des technologies et des outils utilisés ensemble pour construire, développer et déployer une application ou un système) sans remise en cause de l'architecture globale.*	*Refonte ou remplacement des composants développés en interne par des solutions standards du marché. Réduction de la dépendance aux développements spécifiques et de leur coût de maintenance*	*Transformation profonde de la structure applicative (micro-services, cloud-native, orientée évènements, etc.). Changements de paradigme pour répondre aux enjeux de scalabilité, résilience et agilité.*

02

Opportunités de l’IA pour la modernisation des applications

I – Amélioration ponctuelle de la qualité

Description

Corrections ciblées du code existant, et refactorisation limitée sans modification de l'architecture ou des technologies utilisées. Impact localisé sur des modules ou fonctionnalités spécifiques.

Exemples

- Amélioration du code existant
- Correction de *code smell* (mauvaises pratiques de conception)
- Résolution de bug
- Optimisation / Amélioration ponctuelle

×

Méthodes employées

- Prompt engineering simple pour guider l'IA dans la correction ciblée
- Fichiers de contexte limités aux modules concernés
- Retesting régulier
- Validation humaine systématique
- Revue des Golden Rules

×

Outils mobilisés

- **GitHub Copilot** avec modèle premium (Claude Sonnet, etc.) ✓
- **Context7** (*optionnel*) ✓
- **Linters** (SonarCube, Checkstyle, PMD) ✓

Résultats

✓ **Résultats probants** – Les outils mobilisés *(exemples ci-dessus)* sont capables de réaliser ces actions en autonomie, de mettre à jour les tests en fonction et de les rejouer, avec des gains estimés de l'ordre de quelques heures.

II - Amélioration de la sécurité, conformité, et mise à niveau des composants

Description

Mise à jour des bibliothèques, dépendances et composants tiers pour corriger les vulnérabilités. Application de correctifs de sécurité et montée de version des frameworks en place.

Exemples

- Mise à jour de dépendances log4j pour corriger des vulnérabilités connues
- Remplacement de code custom LDAP par l'utilisation d'une librairie Open Source dédiée
- Montées de version (migration log4j 1.x vers Logback, montée de version de Jackson-Databind, etc.)

Méthodes employées

- Déploiement progressif et par étape : d'une version à l'autre, etc.
- Demande de mise à jour des librairies et prise en compte des impacts sur le code existant
- Vérification des métriques qualité/sécurité à chaque étape, idéalement via MCP

Outils mobilisés

- **GitHub Copilot** avec modèle premium (Claude Sonnet, etc.) ✓
- **Context7** ✓
- **Outils d'analyse de la composition logicielle** : Xray, Renovate, etc. ✓

Résultats

Résultats probants – Les outils *(exemples ci-dessus)* sont capables de réaliser ces actions en autonomie, de mettre à jour les tests en fonction et de les rejouer.

III - Modernisation des frameworks et des langages standards

Description

Migration vers des versions récentes des frameworks et langages actuels. Évolution significative de la stack technique (langages, frameworks, bases de données et outils de déploiement nécessaires pour faire tourner le code), sans remise en cause de l'architecture globale.

Exemples

- Upgrade d'un framework OpenSource (Spring Boot 2.x vers Spring Boot 3.x, Angular X vers Angular Y, etc.)
- Migration Java par paliers (6 > 8 > 17 > 21 > 25)
- Remplacement de frameworks legacy (CXF, etc.)

Méthodes employées

- Découpage en sous-tâches par l'expert qui pilote le bot ✓
- Génération par le bot d'un upgrade plan pour diviser le problème en sous-tâches ~
- Modernisation de la stack de test en priorité ✓

Outils mobilisés

- **GitHub Copilot** avec modèle premium (Claude Sonnet, etc.) ✓
- **Github Copilot App Modernization** ~
- **OpenRewrite** (lorsqu'il existe des recettes) ✓

Résultats

✓ **Résultats probants** avec Github Copilot, lorsque l'expert définit un plan d'action en sous-tâches : amélioration de la productivité d'un expert, dont la présence reste essentielle.

~ **Résultats prometteurs mais partiels** avec Github Copilot App Modernization : modernisation incomplète, à n'utiliser que pour des cas simples.

IV - Modernisation des frameworks « maison »

Description

Refonte ou remplacement de composants développés en interne par des solutions plus standards du marché.
Le défi réside dans l'analyse du code existant, la compréhension des fonctionnalités implémentées, et la transposition vers des frameworks standardisés. Ici, l'IA Générative ne connaît pas le framework maison, et doit se baser sur l'analyse du code source et sur des spécifications pour effectuer la migration.

Exemples

- Mise à jour complète d'une application utilisant un framework maison
- Mise à jour du code applicatif pour supprimer les dépendances ou middleware patrimonial « legacy »

Méthodes employées

- Modernisation d'une application manuelle et génération d'un **fichier différentiel** (avant/après) pour guider les futures modernisations
- Utilisation d'un **fichier Markdown « guide »,** qui documente les impacts connus

Outils mobilisés

- **GitHub Copilot** avec modèle premium (Claude Sonnet, etc.)
- **AWS Custom Transform**
- **OpenRewrite** (lorsqu'il existe des recettes)

Résultats

Résultats intéressants :

- Via **fichier différentiel**: permet de produire des recettes OpenRewrite à partir du fichier différentiel
- Via **fichier Markdown « Guide »** : uniquement pour des cas simples pour l'instant, mais des exemples très récents avec des modèles frontières sont probants et s'annoncent prometteurs pour 2026

V – Réécriture

Description

Transformation profonde de la structure applicative (micro-services, cloud-native, orientée événements, etc.). Changements de paradigme pour répondre aux enjeux de scalabilité, résilience et agilité.

Exemples

Migration d'un framework maison vers des technologies modernes (Spring boot, Angular, k8s, ...)

Migration d'une application complète legacy Cobol vers des technologies modernes (Spring boot, Angular, k8s, etc.)

×

Méthodes employées

Spec-Driven Development : rétro-documentation par IA de l'application, puis développement à partir de ces spécifications pivot *(cf. GT 4 Design des applications)*

×

Outils mobilisés

GitHub Copilot avec modèle premium (Claude Sonnet, etc.) ~

SpecKit ~

AWS Q:Developer Transform ~

Kiro (mode Spéc) !

AWS Transform Custom !

Résultats

! **Résultats intéressants** pour AWS Transform Custom, avec le potentiel d'effectuer une majorité du travail de réécriture, sous supervision d'un expert. Kiro est facilitant mais ne peut pas faire une réécriture complète d'application à date.

~ **Résultats partiels** via GH Copilot et SpecKit, qui fonctionnent uniquement avec un contexte de taille modeste.

Des prérequis essentiels pour la modernisation applicative par IA

CODE SOURCE ACCESSIBLE

L'**accès au code source complet** de l'application est indispensable pour permettre à l'IA d'analyser et de modifier le code.

Les approches décrites ici fonctionnent pour des applications allant **jusqu'à 50 000 lignes de code** : au-delà, il est nécessaire de les segmenter.

ENVIRONNEMENT DE DÉVELOPPEMENT

Un **environnement de développement configuré** avec les dépendances nécessaires permet à l'IA de simuler et tester les modifications.

Ceci peut impliquer une **modernisation de l'environnement de compilation** et de gestion des dépendances.

TESTS AUTOMATISÉS

Des **tests (unitaires, d'intégration, de bout en bout) doivent être définis** pour les applications à moderniser. Une étape préalable de création de tests (y compris par GenAI *cf. GT 6 Automatisation des tests)*, peut être nécessaire.

Ces tests doivent être **automatisés** et **exécutables** par l'IA, pour valider les modifications apportées par l'IA et garantir la non-régression.

Les facteurs facilitants pour favoriser la réussite de ces modernisations par IA

Des **bonnes pratiques** ont pu être identifiées à partir des projets de migration réalisés parmi les membres du GT. Celles-ci vont **au-delà de la seule modernisation**, et devront être approfondies en 2026.

Associer IA et outils déterministes et éprouvés

Il est possible d'augmenter les IA en les interfaçant avec des outils « traditionnels » qui ne sont pas basés sur l'IA.

Exemples : la génération de recette OpenRewrite (déterministe) par IA, l'utilisation d'outils type Linter, SonarQube, ArchiUnite, etc.

Laisser l'IA construire elle-même ses instructions…

Les IA peuvent être sollicitées pour générer elles-mêmes les consignes qu'elles doivent suivre, à partir d'exemples (code existant, stratégies de transformation explicites, prompts, etc.).

Exemple : La génération d'instructions génériques de modernisation au format Markdown, à partir du code avant/après d'une application modernisée.

… Et vérifier ses propres productions

Si la validation doit rester humaine, un agent IA peut faciliter la relecture de code généré par un autre. Cette revue doit s'appuyer sur un cadre strict (prompts, outils déterministes, etc.) afin d'éviter des itérations inutiles ou peu pertinentes.

Exemple : l'intégration d'agents relecteurs de code dans la chaîne de CI/CD (Continuous Integration / Continuous Development)

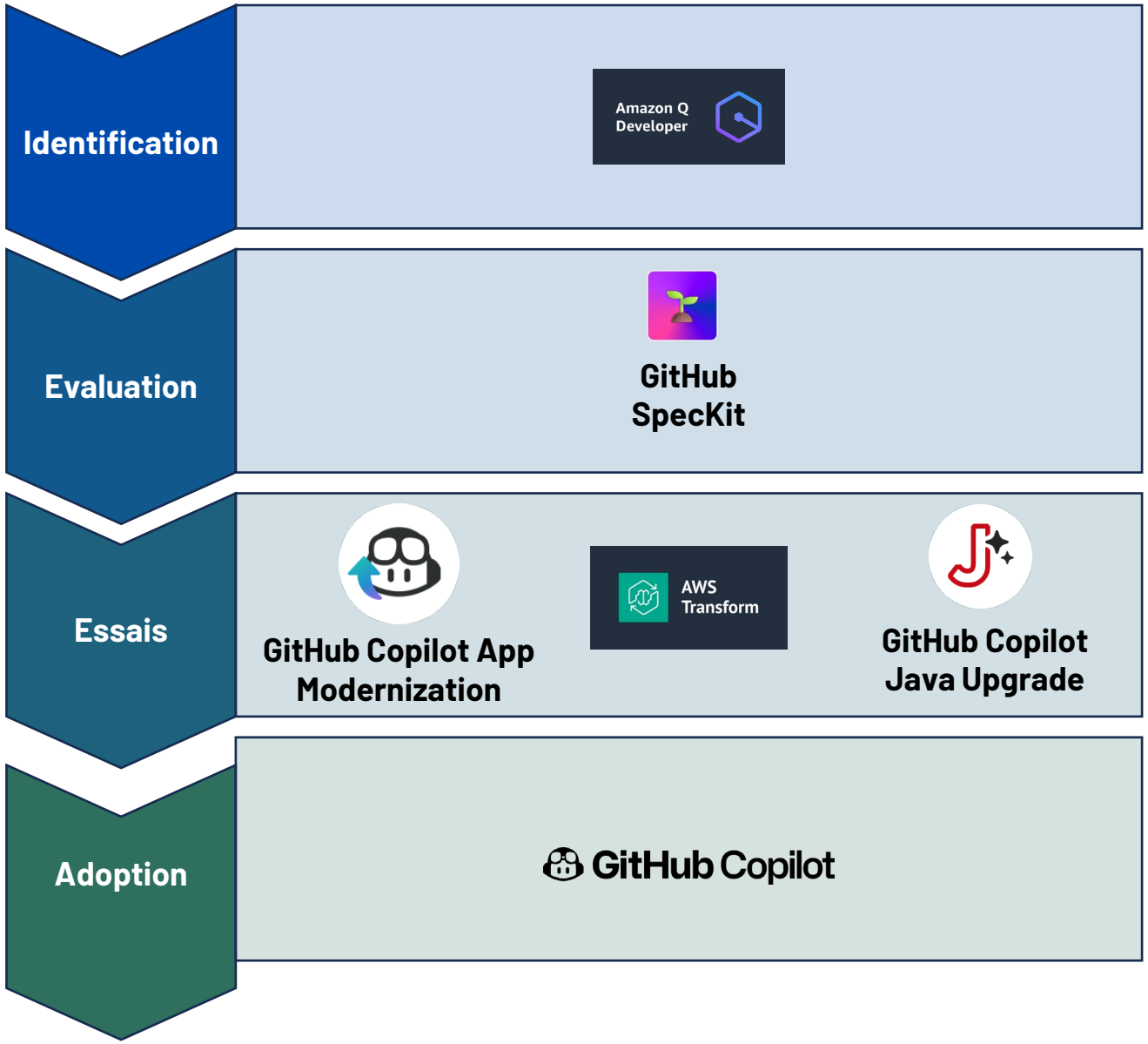
Donner accès à de la « Documentation as Code »

Les performances des IA sont largement augmentées lorsqu'elles ont accès à un contexte à jour de l'entreprise. Ceci invite à gérer la documentation interne comme du code : langages de balisage (Markdown), gestion de version, … Et génération automatique par IA !

Exemple : la création de documentation technique au format Markdown, sa gestion via Git interne et son exposition par MCP aux agents IA.

Positionnement par stade d'utilisation, des outils identifiés par les entreprises participantes

Radar des outils des GT



03

Conclusion

Conclusion

L'IA permet déjà de **répondre complètement aux cas de modernisation « simples » d'application**, et offre une assistance pertinente pour tout ou partie des cas plus complexes.
Les progrès des outils et approches ont été continus en 2025 et ouvrent des **perspectives très prometteuses** : les solutions les plus intéressantes (AWS Custom Transform, Claude Opus 4.5) ont été publiées en décembre 2025 !

De manière générale, adopter une **approche incrémentale de la modernisation d'une application**, autrement dit étape par étape, est fondamental. Cela permet de bien rythmer les vérifications manuelles du code généré selon des groupements cohérents, plutôt que de faire une relecture ligne par ligne très chronophage. Le **mode « Plan »** des agents est à ce titre facilitant, même si celui-ci reste émergent.

La **présence d'un expert**, sauf dans les cas les plus simples *(niveaux 1 et 2)*, reste **obligatoire**. Les agents ont notamment du mal à détecter les fonctionnalités obsolètes ou les cas marginaux.

Sur les **langages les plus anciens** (Cobol/OpenVMS), les résultats ne sont pas vraiment satisfaisants, la rétro-documentation par agent est cependant très efficace.

Ces nouveaux usages offrent par ailleurs une **opportunité d'harmonisation des pratiques de modernisation**, qui sont souvent hétérogènes au sein même de nos sociétés.