Moderniser du code legacy avec Claude Code

Fiche d'étude en français du webinar Anthropic « *Modernizing Legacy Code with Claude Code* » (24/09/2026, 56 min) et de ses 6 documents annexes.
Intervenants : **Belinda Neal** (Managing Director Financial Services), **Harsh Patel** (Applied AI Architect), **Morgan Lunt** (équipe Claude Code, auteur du plugin).
Les repères [mm:ss] renvoient à la vidéo et à la transcription.

> **En une phrase :** les modèles actuels savent comprendre, cartographier et réécrire de gros codes anciens. La difficulté ne vient plus de l'écriture du code mais de **la preuve que le nouveau code se comporte comme l'ancien** et de **l'organisation humaine autour** (règles métier, validation, mise en production). Le plugin open source `code-modernization` structure cette démarche.

1. Déroulé du webinar

Temps	Séquence
00:29 – 06:46	Accueil, sondages auprès des ~2 000 participants, nuage de mots des cas d'usage.
06:46 – 12:26	**Harsh** : pourquoi moderniser maintenant, apport des agents, présentation de Claude Code, d'Opus 5.5 et du plugin.
12:26 – 44:59	**Morgan** : démo complète COBOL → Java Spring (application « CardDemo »), puis démo rapide Java 8 → 17 sur Jetty.
44:59 – 54:02	Questions-réponses : personnalisation, langages, coût, vérification, gestion du contexte.
54:02 – 56:00	Ressources et vision : vers un code « toujours à jour » qui rendrait la modernisation inutile.

Les sondages [01:27–05:53]

- « Avez-vous déjà modernisé du code avec Claude ? » : ~30 % terminé ou en production, 22 % en cours, 26 % prévu, 21 % en exploration.
- Chantiers les plus fréquents : montées de version **Java / .NET**. Environ 30 % des participants ont commencé par **documenter le code et extraire les règles métier**, la base de tout le reste.
- Selon Morgan, il y a un an, ce genre de projet relevait du bricolage expérimental. Aujourd'hui, c'est **prêt pour la production**. Harsh ajoute que la question des clients est désormais : « comment industrialiser ? »

2. Pourquoi moderniser, pourquoi maintenant [06:46]

Les problèmes classiques

- **Perte de savoir** : les auteurs du code sont partis, plus personne ne sait exactement ce qu'il fait.

- **Difficulté de recrutement** sur COBOL et les vieux .NET, donc lenteur pour livrer de nouvelles fonctionnalités.
- **Failles de sécurité** dans des systèmes critiques qui ne sont plus maintenus.
- Les approches passées coûtaient cher : documentation manuelle, consultants à former, « big bang » sur plusieurs années, ou outils de traduction ligne à ligne qui produisent du **« JOBOL »** (du Java qui reste du COBOL dans sa structure, dette technique comprise).

Ce que changent les agents

- Le travail de volume (lire, cartographier, documenter) est délégué aux agents.
- On peut **reconstruire à partir des spécifications retrouvées** au lieu de traduire mot à mot.
- Claude Code parcourt des bases de plusieurs centaines de milliers ou millions de lignes et travaille des heures sans interruption. Le code peut rester chez le client (accès via Bedrock, Vertex ou l'API Anthropic), un point important pour les secteurs régulés.
- **Opus 5.5** (sorti la semaine du webinar) : niveau proche de Fable 5.1 sur la plupart des tâches, pour environ 40 % de coût en moins qu'Opus 5. Un testeur a migré ~700 000 lignes en moins d'une journée.

3. Les trois grands types de modernisation

Harsh les présente, et le *Field guide* (doc 03) les formalise :

Type	Principe	Exemple	Quand le choisir
Uplift	Même techno, version plus récente	Java 8 → 21, C++11 → C++20	La techno convient, mais la version est obsolète (fin de support, failles non corrigées)
Transform	Changement de techno, **comportement identique**	COBOL → Java	La techno est le problème et le comportement actuel est jugé fiable
Reimagine	Reconstruction à neuf, le comportement peut évoluer	Monolithe → microservices	On veut aussi changer le fonctionnel. Il faut alors une spec comportementale écrite et validée

> Le *Field guide* insiste : il faut **trancher ce choix dès le départ**. Les équipes de production veulent en général un *transform* (risque minimal), tandis que les développeurs et le métier poussent vers un *reimagine* (occasion de payer la dette technique et d'ajouter des besoins). Si la question n'est pas tranchée, elle revient plus tard sous la forme de disputes sur la « correction » de chaque changement.

4. La démo COBOL → Java, étape par étape [12:26-40:00]

Terrain de jeu : CardDemo, l'application COBOL open source de référence (gestion de cartes bancaires), avec 44 programmes et 62 copybooks. Morgan choisit volontairement le cas le plus difficile. Il le dit lui-même : un upgrade Java, .NET ou un passage Angular → React est beaucoup plus simple.

Fil rouge : le calcul mensuel des intérêts. Le COBOL **tronque** au lieu d'**arrondir**. Une conversion naïve en Java changerait quelques centimes sur chaque compte, et la comptabilité ne tomberait plus juste. La démo montre comment le processus détecte ce piège.

① Preflight : préparer le terrain

L'agent pose les questions que seul un humain peut trancher :

- Le code fourni est-il complet ? Y a-t-il des dépendances externes ? Si oui, il faut les mettre sur le disque.
- Comment le code se compile-t-il et se teste-t-il ? Quelle CI est en place ?
- Y a-t-il une infrastructure interne (dépôt d'artefacts privé, par exemple) ? Il faudra alors des accès ou un MCP.
- Des tentatives passées ont-elles échoué, et à quel endroit ? C'est précieux pour anticiper les difficultés.
- Des zones sont-elles interdites (raisons de propriété intellectuelle, etc.) ?

Il vérifie ensuite les outils installés (par exemple GnuCOBOL, pour exécuter l'ancien code et comparer plus tard) et détecte **des fichiers manquants** référencés par des `include`. Mieux vaut le savoir avant de commencer.

Au passage, Morgan montre en avant-première les « **Claude Mods** », un mécanisme à venir pour personnaliser l'interface de Claude Code. Il l'utilise pour une barre latérale qui affiche l'avancement.

② Assess : faire l'état des lieux

- Inventaire : fichiers, lignes de code, complexité cyclomatique.
- Architecture, domaines métier, relations entre entités et flux de données, **déduits du code seul** (le README était quasi vide).
- Dette technique et sécurité : mots de passe en clair, import/export cassé, et même **une violation PCI** (norme de sécurité des cartes bancaires).
- Rapport lisible en HTML plutôt que dans le terminal.

> **Message clé de Morgan : « démonter les échafaudages ».** Il y a 6 à 12 mois, il fallait un outillage très spécialisé par type de migration pour compenser les faiblesses des modèles. Aujourd'hui, cet outillage **dégrade les résultats**, parce qu'il bride la capacité de l'agent à résoudre les problèmes lui-même. Son refrain : *« you can just do things now »*. Il ajoute qu'Opus 5.5 lui donne les mêmes résultats que Fable sur ces tâches, pour moitié prix ou moins.

③ Map : cartographier

- **Carte interactive** de toute la base : programmes, stockages de données, liens entre eux. Chaque bloc est dimensionné selon sa taille en code.
- Extraction de **parcours métier / user stories**, par exemple « comment un achat par carte arrive-t-il sur le compte lors du traitement de nuit ? », suivi appel par appel.
- Utile **même sans migrer**, pour comprendre enfin un vieux système hérité.

④ Extract rules : extraire les règles métier

- 323 règles extraites (dont 41 de calcul et 146 de validation de carte), **classées par niveau de confiance**.
- Principe : **économiser le temps humain**. Les règles évidentes ne sont pas soumises aux humains, qui se concentrent sur les cas douteux et les « défauts suspectés ».
- Chaque règle est rédigée en langage clair, avec le COBOL d'origine en regard. Le métier confirme : « oui, cet arrondi est voulu, ne le changez pas ».
- On peut faire circuler une règle vers un expert via Slack (MCP) : la réponse est enregistrée sur disque et le travail continue.
- Le niveau de contrôle humain dépend de la **tolérance au risque** : de « tout vérifier à la main » à « fais au mieux ».

⑤ Brief : le plan à faire valider

- C'est le document « à présenter à son chef » : objectif du pilote, architecture cible, correspondance entre chaque composant ancien et nouveau, points à revoir, séquencement, critères de réussite.
- **Traçabilité** : chaque affirmation cite ses fichiers sources. Morgan conseille de tout versionner dans **Git, avec des commits très fréquents**, pour savoir qui a validé quoi et quand (utile en audit et en conformité).
- L'agent **écrit ses propres outils** (petits scripts Python ou bash, par exemple pour le graphe de dépendances). Le résultat est déterministe et reproductible, et on évite d'envoyer tout le code au modèle pour ça.
- **Pas de big bang.** On commence par un module peu connecté (approche « *strangler fig* », le figuier étrangleur), relié à l'ancien COBOL par des adaptateurs. Pilote retenu : le traitement de nuit et le calcul des intérêts.

⑥ Transform : réécrire le code

- L'agent compile l'ancien et le nouveau code, écrit des tests, puis implémente. Selon Morgan, l'implémentation est **l'étape la plus rapide**.
- Un agent **« architecture critic »** relit le résultat. Il a trouvé une route REST non authentifiée qui permettait de lancer un traitement déplaçant de l'argent. Faille corrigée.

⑦ Verify : prouver l'équivalence

- Pour chacune des 18 règles du module : COBOL d'origine, règle extraite, Java produit et tests qui fixent le comportement, **côte à côte**. Exemple de règle : intérêt = solde × taux / 1200, **tronqué**.
- Des transactions simulées passent dans les deux programmes, avec des sorties identiques, y compris sur les cas d'arrêt anormal attendus.
- Le piège de l'arrondi est signalé et évité. Le système reste à 148 tests passés.
- Résultat : **2 % de la base modernisée**. Ça paraît peu, mais c'est le but : on a eu beaucoup d'occasions de valider la compréhension avant de passer à l'échelle.

> Pourquoi un **plugin** et pas un produit fermé ? Parce que Morgan s'attend à ce que l'échafaudage **devienne inutile** au fil des progrès des modèles. Ce qui restera, c'est le besoin d'injecter le **contexte métier des humains** aux bons moments.

Démo bonus : Java 8 → 17 sur Jetty (~500 000 lignes) [~40:00]

- Parcours **raccourci** : pas d'extraction de règles pour une simple montée de version. On met à jour la version (`pom.xml`, Maven), on regarde ce qui casse, l'agent se corrige lui-même, et on remonte aux humains ce qui est douteux.
- La suite de tests existante sert d'arbitre. L'agent isole 23 différences de comportement à examiner, et le temps humain va uniquement là.
- Coût approximatif selon Morgan : **quelques centaines de dollars** de tokens.

5. Questions-réponses : l'essentiel

Question	Réponse
Comment éviter les régressions ?	En combinant les tests existants, de nouveaux tests écrits par Claude et du **blue/green** (ancien et nouveau en parallèle). Certains font tourner des bases parallèles pendant des semaines pour traquer les écarts. Le bon niveau d'exigence dépend de la tolérance au risque.
Comment adapter le plugin à nos standards ?	Il suffit de **le demander à Claude** : « voici nos conventions et nos skills, modifie le plugin pour les respecter à chaque étape ». Claude Code sait modifier ses propres plugins et skills. Harsh conseille de lui fournir la documentation interne (fichiers .txt) en contexte.
Quels langages sources ?	Morgan n'a pas encore trouvé de langage que le modèle ne sache pas raisonnablement traiter, même des langages ésotériques. COBOL s'est révélé plus facile que prévu.
Comment suivre le coût ?	Créer **une clé API dédiée au projet** et suivre sa facture.
Comment gérer une base plus grosse que le contexte (~1 M tokens) ?	Avec des **« dynamic workflows »** : Claude écrit un script déterministe qui lance de nombreux sous-agents, qui peuvent eux-mêmes en lancer d'autres (façon *map-reduce*). Chacun écrit ses résultats sur disque et remonte un résumé. L'humain ne dialogue qu'avec un seul agent principal.
Comment faire confiance au code produit ?	Comme avec un prestataire humain : on fait tourner les deux versions côte à côte, on cherche les écarts et on itère. Pour un pipeline de données, on rejoue des jeux de données historiques.

Vision de fin [54:02]

Pour Morgan, **écrire du code n'est plus le goulot d'étranglement**. Les goulots sont désormais la validation, la revue et les tests d'intégration. Une fois ces étapes fluidifiées, on pourra garder le code **en permanence à jour** (nouvelle version de Python → mise à jour automatique, tests de non-régression, déploiement). La « grande modernisation » deviendrait alors un sujet du passé.

6. Ce qu'apportent les documents annexes

01–02 · Le plugin `code-modernization`

Installation : `/plugin install code-modernization@claude-plugins-official`. Point d'entrée : `/code-modernization:modernize`, qui pose quelques questions puis indique la commande suivante. **Le code legacy n'est jamais modifié** : le plugin écrit uniquement dans `analysis/<nom>/` et `modernized/`, et met à jour un rapport `REPORT.html` unique.

Étape	Commande	Résultat
0	`modernize`	Recueille l'intention (`INTENT.md`)
1	`modernize-preflight`	Environnement prêt ? 5 questions humaines, build vérifié, sources manquantes
2	`modernize-assess`	Inventaire, complexité, dette, sécurité, approche recommandée
3	`modernize-map`	Carte interactive (dépendances, flux, parcours métier)
4 / 4b	`modernize-extract-rules` / `-review`	Règles au format *Given/When/Then* avec référence `fichier:ligne`, revérifiées par un second agent, puis validées par un humain
5	`modernize-brief`	Plan par phases. **Rien n'est construit sans approbation**
6	`uplift` / `transform` / `reimagine`	La réalisation
7	`modernize-verify`	La preuve : vérification indépendante, idéalement dans une nouvelle session, avec un verdict par module
8	`modernize-harden`	Scan de sécurité de l'ancien système, avec un correctif à appliquer soi-même

À tout moment, `modernize-status` indique où on en est. Le plugin identifie **6 décisions réservées aux humains** : les réponses au preflight, la revue des règles douteuses, l'approbation du brief, l'acceptation des écarts, la signature de la preuve et l'application du correctif de sécurité. Les dossiers `commands/` et `agents/` du dépôt se lisent comme de la documentation.

03 · Field guide : préparer l'organisation (le document le plus utile)

Idée centrale : l'agent accélère l'écriture, donc **le goulot se déplace vers l'organisation** (revue, validation, mise en production). Le guide propose 6 étapes :

1. **Définir la cible** : type de modernisation (uplift, transform ou reimagine), cartographie, spec comportementale, justification. Le premier bénéfice est souvent la **réduction du risque**, bien plus que la baisse des coûts.
2. **Définir le « certificat »** : l'ensemble des conditions, vérifiables sans humain, qu'un changement doit remplir. Par exemple : tests d'origine et nouveaux tests qui passent, couverture minimale, performances tenues, revues adverses par Claude dans un contexte vierge, sorties identiques pour des entrées identiques, aucune nouvelle alerte de sécurité. Il se rédige **avec ceux qui valideront**.
3. **Fixer la « politique de promotion »** : un circuit de revue à plusieurs niveaux, selon l'impact du changement et la confiance de l'agent. On fait intervenir les experts **tôt**

(l'inverse du schéma classique, où la revue arrive à la fin). La décision doit venir de la direction, pour que la responsabilité soit partagée.

4. **Mettre en place les prérequis** : machine dédiée, capacité de test, CI/CD, gel du code éventuel, droits minimaux (pas d'accès à la production), secrets et données personnelles masqués, traçabilité de chaque changement.
5. **Construire et affiner le workflow agentique** (en partant du plugin). Quand un problème apparaît, **on corrige le workflow, pas chaque changement**.
6. **Exécuter** : un petit périmètre de bout en bout d'abord, puis passage à l'échelle. Mesurer les tokens du pilote pour extrapoler le budget.

04 · Blog « migrations à grande échelle » (Zig → Rust)

- **Bun** (Jarred Sumner) : environ 1 million de lignes Zig → Rust en **moins de deux semaines**, avec 100 % des tests existants au vert avant fusion, puis 19 régressions trouvées et corrigées. Coût : environ **165 000 $** au tarif API.
- **Mike Krieger** : un code Python porté en 165 000 lignes de TypeScript en un week-end. La compilation est passée de ~8 min à ~2 s.
- Formule clé : **on ne corrige pas le code, on corrige le processus (la boucle) qui l'a produit**.
- Prérequis indispensable : un **« juge » solide**, c'est-à-dire des tests capables d'évaluer l'ancien et le nouveau code sur un pied d'égalité.
- Pourquoi ça marche bien : le travail est parallélisable, l'ancien code sert de spec, les tests servent d'arbitre et chaque échec devient la tâche suivante dans la file.
- Le pire scénario n'est plus un projet de quatre ans raté : **on supprime la branche et on recommence**.

05 · Code Modernization Playbook (eBook)

Document plus ancien (il cite Sonnet 4 et Opus 4.1) et davantage orienté décideurs. Il aide à **construire le business case** : les trois freins (pénurie de compétences legacy, baisse de productivité, dette technique et sécurité), des cas d'usage, le calcul du ROI, le choix d'un outil agentique et une démarche de lancement. Utile pour argumenter auprès d'un client, moins pour la technique.

06 · Cas client LG CNS

Modernisation de systèmes d'entreprise vieux de 20 ans : environ 2 900 API migrées avec ~99 % de réussite, une migration de plusieurs millions de dollars bouclée en 7 mois pour environ la moitié du coût d'une refonte classique, et plus de 1 300 écrans migrés vers React. Claude Code est devenu le standard pour leurs ~200 ingénieurs. C'est une illustration concrète du discours du webinar.

7. À retenir pour toi

1. **Commencer par comprendre.** Le trio `assess` + `map` + `extract-rules` vaut le coup même sans migrer : il documente un vieux projet hérité (client ou perso) en peu de temps.
2. **Le vrai livrable, c'est la preuve**, pas le code : tests, comparaison ancien/nouveau, traçabilité dans Git.
3. **Le temps humain est rare.** Il faut le réserver aux décisions métier et aux cas douteux, le reste va aux agents.
4. **Petit pilote d'abord** (un module peu connecté), mesure du coût, puis passage à l'échelle.
5. **Ne pas trop encadrer le modèle.** Les garde-fous d'il y a six mois peuvent brider les modèles actuels.
6. **Argument commercial en freelance** : un audit ou une cartographie de code legacy, livré sous forme de rapport HTML et de carte interactive, est désormais une prestation rapide à produire et facile à vendre.

8. Petit glossaire anglais → français

Terme	Sens
Legacy code	Code hérité, ancien, souvent peu documenté
Uplift / Transform / Reimagine	Montée de version / changement de techno à iso-fonctionnel / reconstruction à neuf
Business rules	Règles métier (ce que le logiciel doit faire, vu de l'utilisateur)
Copybook	Fichier COBOL de définitions partagées (≈ un fichier d'en-tête / include)
Preflight	Vérifications préalables avant de lancer le chantier
Brief / Bill of materials	Plan de projet synthétique à faire valider / inventaire de ce qui sera fait
Strangler fig	« Figuier étrangleur » : remplacer un système morceau par morceau, en le laissant tourner
Shim	Adaptateur qui fait dialoguer l'ancien et le nouveau code
Blue/green	Deux versions déployées en parallèle pour comparer et basculer sans risque
Equivalence proof / parity	Preuve que le nouveau code se comporte comme l'ancien
Characterization tests	Tests qui « figent » le comportement actuel, même bizarre, pour le préserver
Certificate	(Field guide) Liste des conditions qu'un changement doit remplir pour être accepté
Promotion policy	(Field guide) Circuit de validation d'un changement jusqu'à la production
Adversarial review	Relecture critique par un agent indépendant chargé de trouver les failles
Dynamic workflow	Script écrit par Claude qui orchestre de nombreux sous-agents en parallèle
Scaffolding / harness	Échafaudage / outillage qui encadre le modèle
SME (Subject Matter Expert)	Expert métier du domaine
YOLO	« On fonce sans filet » (lancer la migration sans étapes de contrôle)
Truncate vs round	Tronquer vs arrondir : le piège des centimes de la démo
PCI (DSS)	Norme de sécurité des données de cartes bancaires

Fiche rédigée le 26/09/2026 à partir de la transcription anglaise officielle (Goldcast) et des documents de l'onglet « Docs » du webinar. Les chiffres cités proviennent des intervenants ou des documents eux-mêmes.