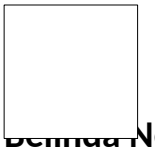

Belinda Neal
00:29 - 01:00

Bonjour. Bon après-midi.

Bonsoir. Bienvenue à notre webinaire d'aujourd'hui consacré à la modernisation du code hérité avec Claude Code.

Nous sommes ravis de vous compter parmi nous. Merci de nous consacrer un peu de votre temps, et nous avons hâte de commencer la session et la démonstration d'aujourd'hui.

Nous aurons du temps pour les questions, ce qui est très réjouissant. Mais avant toute chose, nous allons nous présenter : voici vos intervenants du jour.

Pour me présenter brièvement, je m'appelle Belinda Neal. Je suis directrice générale des services financiers ici, chez Anthropic.

Mon travail se situe à la croisée des services financiers et de notre secteur, et j'ai vraiment hâte de participer à la discussion d'aujourd'hui. Harsh, je te laisse la parole pour te présenter.

Harsh Patel
01:00 - 01:15

Bonjour à tous. Je suis ravi de vous rencontrer ici.

Je m'appelle Harsh. Je fais partie de l'équipe d'IA appliquée ici chez Anthropic.

Dans le cadre de mes fonctions, je travaille en étroite collaboration avec bon nombre de nos entreprises de services financiers pour les aider à déployer Claude et à tirer le meilleur parti de ses capacités. Morgan, à vous.

Morgan Lunt
01:15 - 01:27

Salut à tous. Je m'appelle Morgan.

Je travaille au sein de l'équipe de développement de Claude. Je suis particulièrement passionné par la modernisation du code et la collaboration avec les grandes entreprises.

Je suis donc ravi de vous parler aujourd'hui de certaines expériences que j'ai vécues dans ce domaine.

Belinda Neal
01:27 - 03:43

Merci beaucoup. Très bien.

Passons en revue l'ordre du jour d'aujourd'hui. Tout d'abord, nous allons parler des raisons pour lesquelles il faut moderniser dès maintenant et pourquoi choisir Program Store.

Et, bien sûr, c'est une semaine très passionnante pour nous ici chez Anthropic avec la sortie de notre tout dernier modèle, Opus 5. 5.

Ensuite, nous allons voir comment les agents peuvent transformer la modernisation. Puis, nous allons passer en revue le plugin de modernisation du code et assister à une démonstration en direct.

Ensuite, nous aborderons quelques exemples de réussite et les premières étapes à suivre, puis nous consacrerons un peu de temps à une séance de questions-réponses à la fin. Si vous souhaitez commencer à soumettre vos questions pour le webinaire, n'hésitez pas à le faire ; nous prendrons le temps de les traiter en priorité vers la fin.

Quelques informations pratiques pour commencer. Tout d'abord, un enregistrement de cette session vous sera envoyé par e-mail dans les prochaines vingt-quatre heures.

Si vous devez manquer une partie de la session ou si vous souhaitez la revoir, vous pourrez donc y accéder. Deuxièmement, nous venons d'évoquer les questions.

N'hésitez pas à en ajouter tout au long des démonstrations et de la discussion. Enfin, les retours d'expérience.

Merci de bien vouloir remplir le questionnaire de satisfaction à la fin de la session. Nous serions ravis de connaître votre avis, de recevoir vos questions et vos suggestions pour mieux vous impliquer à l'avenir.

Très bien. Pour commencer, nous aimerions poser quelques questions au public.

Nous avons environ 2 000 participants au webinaire en ce moment. Commençons donc par la première question.

Avez-vous modernisé du code avec Claude ? Prenons quelques minutes pour laisser le temps aux participants de répondre à cette question. D'accord.

Affichons les résultats et voyons ce que les participants ont répondu. Je suis vraiment impatient de découvrir leurs réponses.

Et Harsh, Morgan, je suis également très curieux d'entendre vos points de vue sur les conversations avec les clients aujourd'hui. C'est certainement un phénomène que l'on observe beaucoup dans les services financiers et, espérons-le, plus largement dans l'ensemble de l'écosystème.

Donne-leur encore trente secondes. On dirait que les réponses sont un peu mitigées pour l'instant, c'est donc assez intéressant.

D'accord. On dirait que les résultats sont assez homogènes.

Je pense donc que les résultats sont en fait assez mitigés. Il semble que certaines personnes aient réellement fait des progrès dans ce domaine, avec environ 30 % de projets achevés ou en production.

Vingt-deux pour cent d'entre vous sont en cours de mise en œuvre. C'est encourageant.

26 % ne l'ont pas encore fait, mais cela figure dans votre feuille de route. C'est excellent.

J'espère que ce webinaire d'aujourd'hui vous aidera à acquérir les compétences et les outils nécessaires pour y parvenir. Enfin, 21 % d'entre vous en sont encore au stade de l'exploration.

C'est le moment idéal pour commencer. Alors, Harsh, Morgan, avez-vous des observations ou des commentaires à faire à la lumière de ces résultats ?

Morgan Lunt
03:43 - 04:10

Je dirais ceci : cela correspond à ce que j'ai pu constater lors de mes échanges avec les clients.

Je veux dire par là que nous en sommes désormais à un stade de maturité intermédiaire, où il est possible de mener une modernisation du code à grande échelle avec Claude. Il y a un an, c'était encore assez expérimental : on essayait de bricoler pour voir ce qu'on pouvait faire.

Aujourd'hui, la solution est vraiment prête pour les charges de travail en production et pour des migrations de grande envergure. Nous commençons à constater que nos clients obtiennent d'excellents résultats sans avoir à recourir autant à des solutions de fortune qu'ils auraient dû le faire il y a six mois ou un an.

Harsh Patel
04:10 - 04:25

Tout à fait d'accord.

Et, vous savez, avec les projets pilotes en cours, une autre question que nos clients nous posent très souvent est la suivante : une fois que nous nous sommes lancés, comment industrialiser réellement cette solution à grande échelle ? N'est-ce pas ? Nous allons donc aborder certains de ces aspects dans cette présentation.

Belinda Neal
04:25 - 05:11

Excellent. C'est une excellente base.

Très bien. Passons maintenant à notre deuxième sondage, qui vise à aborder plus précisément les domaines sur lesquels vous avez réellement concentré vos efforts.

Nous allons donc afficher le deuxième sondage, qui vous demandera : « Parmi ces initiatives, lesquelles avez-vous mises en œuvre ? » Si vous avez déjà mis en œuvre une initiative ou si vous en avez une en cours, n'hésitez pas à répondre, car nous sommes très curieux de savoir sur quels domaines vous vous êtes concentrés jusqu'à présent. Cela nous donnera également un aperçu du profil de l'auditoire et des thèmes sur lesquels vous souhaitez vous concentrer aujourd'hui. Laissons donc ce sondage en ligne quelques instants et voyons ce que les participants nous répondent.

Très bien. Nous allons laisser le sondage ouvert pendant quelques minutes.

Les résultats semblent assez cohérents avec ce que nous avons entendu jusqu'à présent.

Harsh Patel
05:11 - 05:19

Il y a beaucoup de mises à niveau vers Java ou .NET. Cela correspond bien à ce que nous entendons également de la part de nombreux clients.

Belinda Neal
05:19 - 05:33

Excellent. Je pense que la documentation du code hérité et l'extraction des règles métier constituent sans aucun doute une base essentielle ; c'est donc très encourageant de constater que les réponses à ce sujet s'élèvent jusqu'à présent à environ 30 %.

Je pense que c'est probablement par là que beaucoup de gens commencent. Morgan, souhaitez-vous ajouter quelque chose à ce sujet ?

Morgan Lunt
05:33 - 05:53

Oui. Je veux dire, nous aborderons un peu ce sujet lors de la démonstration, mais comprendre ce que fait votre ancien code n'est pas toujours évident, en particulier pour les codes plus anciens de type mainframe.

C'est pourquoi il est toujours essentiel de documenter ces règles, d'extraire la logique métier et de comprendre ce qu'il fait avant de se lancer dans sa réimplémentation. C'est là que vous devrez impliquer vos parties prenantes métier.

Belinda Neal
05:53 - 06:46

Parfait. Eh bien, merci beaucoup d'avoir répondu à cette question.

Je pense que nous avons une bonne base de référence sur ce que nous avons abordé et j'espère que nous pourrons partager ici quelques éléments, comme la modernisation de COBOL aujourd'hui, qui permettront aux participants d'y voir un peu plus clair. D'accord.

Revenons aux diapositives. Que voyons-nous donc ? Nous avons créé un petit nuage de mots ici, simplement pour donner à ce groupe un peu d'inspiration sur ce que les gens commencent à faire avec Claude Code en matière de modernisation du code.

De toute évidence, il existe un grand nombre de cas d'utilisation spécifiques que les gens testent et utilisent jusqu'à présent, et nous tenions vraiment à les partager comme point de départ pour aider chacun ici présent à réfléchir à ce qui serait pertinent pour ses feuilles de route et à ce qu'il envisage pour l'instant. J'espère donc que nous aborderons certains d'entre eux aujourd'hui.

Nous ne pourrons évidemment pas tous les aborder, mais, encore une fois, il existe de nombreuses possibilités d'appliquer Claude Code à vos projets de modernisation. Sur ce, Harsh, je vous passe la parole et nous allons commencer par les raisons qui sous-tendent la modernisation du code.

Harsh Patel
06:46 - 12:26

Parfait. Merci beaucoup, Belinda.

Avant de passer à la démonstration proprement dite, je voudrais prendre quelques minutes pour revenir sur ce que nous constatons sans cesse lorsque nous discutons avec nos clients. Bon nombre d'entre vous ici présents sont sans doute des responsables techniques travaillant dans le domaine des produits ou dans d'autres domaines similaires.

Et ce que nous entendons souvent, c'est que bon nombre des systèmes critiques sur lesquels reposent vos activités s'appuient sur des bases de code vieilles de plusieurs années, voire de plusieurs décennies. L'une des conséquences de l'écoulement du temps depuis l'écriture de ce code est une importante perte de savoir-faire institutionnel, car bon nombre des personnes qui l'ont initialement développé ont depuis longtemps quitté l'entreprise ou changé de poste.

Concrètement, cela se traduit ainsi dans vos feuilles de route produit : non seulement cela rend difficile, voire impossible, la mise sur le marché de nouveaux produits répondant aux demandes de vos clients, mais il est également très difficile de recruter du personnel capable de travailler avec des frameworks hérités tels que COBOL ou des versions plus anciennes de .NET, ou tout autre framework de ce type, afin de pouvoir développer très rapidement de nouvelles fonctionnalités. Enfin, bien sûr, il y a le fait que bon nombre de ces systèmes critiques peuvent rester exposés, ce qui peut entraîner une multitude de vulnérabilités susceptibles de présenter des risques de sécurité.

Historiquement, lorsque vous lanciez ces initiatives de modernisation par le passé, vous deviez souvent recruter ou mobiliser d'importantes ressources en interne pour documenter ou extraire manuellement la logique métier, ce qui prenait beaucoup de temps. Vous deviez également « réhydrater » ces cabinets de conseil externes en leur fournissant toutes les informations et connaissances nécessaires.

Un autre risque majeur réside dans le fait que, si vous deviez procéder à une refonte complète, vous auriez peut-être prévu plusieurs années d'expertise ou de temps pour cette tâche spécifique. De plus, bon nombre des outils d'origine dont vous disposez actuellement se contentent, par exemple, de traduire le COBOL en Java sans réduire la dette technique, c'est-à-dire le JOBAL.

Ce sont là bon nombre des défis que nous rencontrons sans cesse. Mais c'est précisément ce qui rend les agents vraiment, vraiment efficaces.

Vous pouvez déléguer une grande partie du travail de volume à ces agents pour qu'ils lisent, cartographient et documentent le code en s'appuyant sur vos

compétences organisationnelles, et qu'ils le restituent dans un style qui correspondra le mieux aux initiatives sur lesquelles vous travaillez. De plus, au lieu de vous contenter de traduire du COBOL vers Java ou d'effectuer une conversion DOBOL, vous pouvez véritablement construire de zéro à partir des spécifications récupérées dont vous disposez.

C'est précisément là qu'un code génératif accélère véritablement les programmes de modernisation. Et pour ceux d'entre vous qui ne connaissent peut-être pas encore Claude Code, il s'agit d'un outil de codage agentique intégré à notre terminal.

Nos équipes produit et ingénierie travaillent d'arrache-pied pour tirer le meilleur parti des capacités de Claude. Ainsi, dans le cadre de ces programmes de modernisation du code, il est capable d'effectuer des recherches efficaces dans de vastes bases de code, qu'elles comptent des centaines de milliers, voire des millions de lignes de code, tout en fonctionnant sans interruption pendant de nombreuses heures d'affilée.

Et pour beaucoup d'entre vous qui venez peut-être de secteurs ou d'institutions réglementés, le principal avantage réside dans le fait que votre code reste sur site et que vous pouvez vous connecter et être opérationnel en utilisant d'autres solutions telles que Bedrock, Vertex ou notre plateforme. De plus, comme Belinda l'a mentionné au début, nous venons de lancer en début de semaine notre dernier modèle, Claude Opus 5.

5. Ce qui est vraiment passionnant, c'est qu'il offre des performances équivalentes à celles de Claude Fable 5.

1 pour la plupart des tâches, tout en coûtant environ 40 % moins cher à faire fonctionner que l'Opus 5. Pour citer quelques chiffres clés que nous avons mentionnés dans notre article de lancement, un des premiers testeurs a réussi à migrer environ 700 000 lignes de code en moins d'une journée, ce qui démontre la puissance de ce modèle pour bon nombre de ces tâches de codage automatisées.

Mais, de plus, pour ces problèmes de longue haleine sur lesquels vous travaillez peut-être, nous réduisons les lectures de cache de 60 %, ce qui diminue considérablement le coût d'exécution de ces charges de travail. À ce stade, vous vous demandez peut-être : « Comment puis-je me lancer ? », n'est-ce pas ? Morgan Lunt a écrit ce plugin de modernisation du code, disponible en open source, qu'il présentera juste après.

Ce plugin met en place un workflow qui ne nécessite pas de lire de documentation : il suffit de le pointer vers l'un de vos dépôts hérités, et vous pouvez considérer qu'il vous guide à travers trois phases principales. La première phase consiste à extraire la logique métier présente dans votre code, à cartographier les dépendances sous forme visuelle, mais aussi à vous fournir une évaluation et à établir des liens avec, par exemple, vos pipelines CICD, vos dépôts Git, etc., afin de déterminer si votre environnement est prêt ou non.

À l'issue de ce processus, l'outil vous fournira un résumé ou une nomenclature que vous pourrez ensuite valider, après quoi vous pourrez emprunter l'une des trois voies possibles. De nombreux clients avec lesquels nous discutons souhaitent simplement, par exemple, procéder à une mise à niveau.

Ils peuvent par exemple passer d'une ancienne version de Java, comme Java 8, à Java 21, ou bien ils peuvent souhaiter procéder à une décomposition d'une architecture monolithique, c'est-à-dire transformer une application monolithique en microservices. Enfin, si vous souhaitez repartir de zéro, vous pouvez utiliser la première phase pour identifier l'ensemble de la logique métier et repartir sur de nouvelles bases grâce à la phase de « réinvention ».

Sur ce, je vais passer la parole à Morgan, qui va vous présenter une démonstration passionnante de l'utilisation de ce plugin de modernisation du code, à travers un exemple concret. Morgan, à vous.

Morgan Lunt

12:26 - 44:59

Très bien. Merci beaucoup, Harsh.

Je vais afficher mon écran pour que vous puissiez tous le voir. Voilà.

Vous devriez donc pouvoir voir mon terminal ici. Avant de commencer, je tiens à préciser qu'il s'agit d'un plugin que j'ai créé pour condenser toutes les connaissances que j'ai acquises au fil des années en matière de modernisation de code.

Je ne vais pas prétendre qu'il existe une seule et unique bonne façon de moderniser votre code. Ce n'est certainement pas le cas.

Et je vous encourage à l'essayer, à le peaufiner, à le modifier, à vous l'approprier. Il est entièrement open source.

Nous avons un lien vers ce plugin, que nous allons partager. L'idée ici, cependant, est d'intégrer autant que possible les meilleures pratiques pour comprendre votre code actuel, extraire les règles métier et les comportements, effectuer des validations après la transformation du code, et tout ce qui touche à la transformation proprement dite du code afin de vous donner l'assurance qu'il va effectivement se comporter correctement et résoudre le problème métier que vous souhaitez voir résolu.

Vous pouvez toujours simplement dire à Claude : « YOLO, convertis ça de C en Rust. S'il te plaît, ne fais aucune erreur. »

Et, franchement, de nos jours, il ne s'en sortira pas si mal que ça. Mais le but ici est de montrer aux clients issus de secteurs réglementés, ou à ceux qui exigent un niveau de confiance plus élevé dans leur code, les types de démarches que vous pouvez entreprendre pour utiliser Claude de manière structurée dans le cadre de votre projet de modernisation du code.

J'ai essayé d'intégrer tout cela dans le plug-in, afin que vous puissiez tous l'essayer, le remixer, le modifier, et vraiment vous l'approprier pour qu'il réponde à vos besoins. Alors, allons-y.

Je vais donc exécuter ici mon plug-in de modernisation, et tout d'abord, voyons sur quoi nous travaillons. Il s'agit d'un programme COBOL, une démo « card », qui est en quelque sorte la norme de facto pour le COBOL open source.

Je me concentre ici sur le COBOL, car je cherche délibérément à illustrer une transformation entre un code source très complexe et très différent et une cible très différente ; dans ce cas précis, il s'agit de passer du COBOL à Java. Si vous essayez de mettre à niveau un runtime Java ou .NET, ou de passer d'Angular à React, par exemple, votre vie sera bien plus simple.

Et après la démonstration COBOL, je pourrai vous en montrer une très rapide en Java. Ce sont des problèmes bien plus faciles à résoudre.

Nous commençons donc par un cas vraiment difficile qui nécessitera un peu plus d'intervention humaine, simplement pour vous donner une idée de la manière dont vous pouvez vous y prendre, et vous pourrez supprimer les étapes qui vous semblent inutiles. Pour vous donner une idée de l'application sur laquelle nous travaillons ici, elle comporte 44 programmes COBOL et 62 « copybooks ».

Nous allons passer en revue chacune de ces étapes : la vérification préalable, la compréhension de votre environnement, la vérification que vous pouvez effectivement compiler le code et que vous disposez de toutes les dépendances

nécessaires sur votre disque, l'évaluation du contenu, la cartographie de l'architecture du code existant et des règles métier qu'il contient, afin de nous assurer que le comportement du code de transformation final correspondra à celui du code d'origine. Nous allons établir un plan.

Nous allons le transformer. Nous allons vérifier qu'il fonctionne et vous montrer où nous en sommes actuellement.

Un point à suivre est cette tâche mensuelle de calcul des intérêts que vous allez voir ici. Le code COBOL tronque les nombres.

Il n'arrondit pas, et c'est donc un comportement qui, si nous nous contentons de le transformer naïvement en Java, va poser problème. Je vais vous montrer comment nous détectons cela dans le cadre de ce workflow. Revenons donc ici : nous avons notre code COBOL, nous chargeons le code Claude, nous lançons le plugin de modernisation du code et nous lui indiquons que nous souhaitons migrer vers Java Spring.

Il va donc commencer par exécuter la pré-validation, et il va me poser quelques questions simplement pour essayer de comprendre, grâce à une personne qui connaît cette base de code – du moins, je l'espère, peut-être un peu –, quelles informations pourraient être utiles à l'agent et qui ne sont pas déjà présentes dans le code qu'il va pouvoir trouver par lui-même. S'agit-il donc du système complet ? Y a-t-il des dépendances externes ? S'il y en a, il vous demandera probablement de les copier sur le disque afin de pouvoir les examiner, dans la mesure du possible.

Et si ce n'est pas possible, vous savez, il vaut mieux qu'il en soit informé, plutôt que de chercher à l'aveuglette. Il voudra savoir à quoi ressemble votre infrastructure de CI et quelle est votre approche en matière de tests de build.

Encore une fois, avec un langage comme Java, c'est très simple. Vous pouvez très fréquemment compiler votre code et le valider.

Pour d'autres langages, cela prend plus de temps, et vous souhaiterez peut-être adopter une approche différente ou regrouper davantage de modifications avant de passer aux étapes de vérification. Nous allons donc répondre à ses questions ici.

Dans ce cas précis, nous avons installé la nouvelle version de COBOL sur cet environnement ; il pourra donc exécuter une partie de ce code COBOL d'origine pour nous permettre d'effectuer des tests de vérification « blue-green » vers la

fin. Et il vous demandera des informations sur l'infrastructure propriétaire et d'autres éléments spécifiques.

Par exemple, si vous disposez d'un Artifactory propriétaire hébergeant vos dépendances, l'outil voudra en être informé et vous donnera la possibilité de fournir des identifiants, un MCP ou tout autre élément nécessaire pour récupérer ces informations supplémentaires, afin qu'il puisse effectuer correctement son travail. Et puis, s'il y a eu des tentatives antérieures de modernisation, si quelqu'un a déjà essayé et échoué, s'il y a des résultats partiels, s'il y a un point de blocage dont vous avez connaissance, du genre : « On a engagé un cabinet de conseil pour essayer de faire ça, et ça a capoté à ce stade. »

Et puis nous avons réessayé il y a quatre ans, et ça a échoué à ce stade-là. Ce sont des informations utiles que l'agent doit connaître.

On peut ainsi essayer d'anticiper les difficultés potentielles et les domaines où l'outil pourrait avoir besoin de plus d'informations de la part des humains. Encore une fois, plus on lui fournit d'informations, plus il sera susceptible de nous aider.

J'ai donc répondu aux questions préliminaires ici, et le système va commencer à inspecter les fichiers proprement dits. Dans ce cadre, il veut savoir s'il y a des zones interdites. Par exemple : ce dossier hérité est-il quelque chose que je ne dois absolument pas toucher et sur lequel je dois travailler dans un autre répertoire ? C'est parfois le cas, notamment pour des raisons de propriété intellectuelle ; nous avons des clients qui ont ce genre d'exigences.

L'outil se met donc au travail. Et vous pouvez voir ici que ce plugin utilise une toute nouvelle fonctionnalité que nous allons bientôt lancer, appelée « Claude Mods ».

Je vous en donne à tous un petit aperçu en avant-première. En résumé, il s'agit d'un moyen extrêmement extensible de personnaliser pratiquement tout ce qui concerne l'environnement Claude Code, y compris l'interface utilisateur.

Dans ce cas précis, j'ai donc un mod implémenté via le plugin de modernisation du code pour ajouter cette barre latérale très pratique qui va nous permettre de garder le cap et de suivre notre progression tout au long de ce processus de modernisation du code. Maintenant que la pré-validation est terminée, tout est identifié pour moi, d'accord.

Je suis désormais prêt à évaluer votre code, à le cartographier, à en extraire des règles, à construire un graphe de dépendances pour vous, simplement pour mieux cerner ce code avant même d'envisager de le transformer. Il va donc me

recommander, pour commencer, un rapport de pré-vérification, en fonction des réponses que je lui ai fournies manuellement ici : la portée de ce qu'il est censé examiner, les répertoires contenant le code à transformer, ainsi que l'état actuel.

Par exemple : quels outils sont installés sur cette machine ? Il va recommander des outils supplémentaires s'il estime en avoir besoin. Par exemple, si je n'avais pas installé la nouvelle version de COBOL ici, il m'aurait probablement recommandé de le faire afin de pouvoir exécuter le code COBOL d'origine à des fins de validation plus tard dans le processus.

Il va s'assurer de bien comprendre comment procéder à la compilation. La chaîne d'outils est présente et, là encore, il vous aidera à vous procurer tout cela si vous ne l'avez pas déjà.

Il dispose d'informations sur la structure du programme COBOL lui-même, les bibliothèques d'exécution qu'il utilise, la chaîne d'outils de compilation et l'exhaustivité du code source. Et dans ce cas précis, il a remarqué : « Tiens.

Il manque en effet certains fichiers. Certaines instructions « include » font référence à des fichiers qui ne se trouvent pas sur le disque.

Vous devriez probablement essayer de les retrouver. Si vous ne les trouvez pas, l'outil contournera le problème.

Mais c'est le genre de problèmes qu'il vaut mieux identifier dès le départ, avant que votre agent ne se lance et n'essaie de travailler avec ce dont il dispose. Nous sommes donc prêts à commencer.

Quelle est la prochaine étape ? Nous allons lancer notre évaluation. C'est là que l'outil commence à examiner en profondeur les fichiers de code pour voir ce qu'ils contiennent.

Il va rechercher des failles de sécurité. Il va rechercher les problèmes de conformité qu'il pourrait détecter, des éléments dont vous voudrez probablement avoir connaissance au fur et à mesure que vous avancez dans le processus de modernisation, afin de vous assurer que votre code traduit ne présente pas les mêmes problèmes que votre code d'origine.

Encore une fois, se contenter d'une transformation mécanique, un peu bête, « un pour un », n'est pas exactement ce que vous souhaitez dans 95 % des cas. L'outil vérifie également la présence de mots de passe et de secrets dans le code qui n'ont rien à y faire.

Et comme précédemment, il me fournit un petit rapport très pratique, car même si j'adore le terminal, je trouve qu'une page HTML est un peu plus lisible. Il s'en chargera donc pour vous.

Les agents sont désormais très performants dans ce genre de tâches. Nous disposons donc ici d'un inventaire un peu plus complet du système : tous les différents fichiers, le nombre de lignes de code, leur taille, la complexité cyclomatique… autant d'informations utiles pour quantifier la complexité de cette transformation.

L'architecture a été extraite. Le code que je lui ai fourni comportait en quelque sorte un petit fichier « Lisez-moi ».

La documentation était très succincte. L'outil va donc déduire du code lui-même la structure, les domaines métier, les relations entre entités et les flux de données.

Il déduit tout cela à partir du code. Les modèles sont désormais capables de faire ce genre de choses.

Il y a six mois, il y a un an, lorsque je travaillais sur des projets de modernisation comme celui-ci, il fallait construire autour des modèles un « harnais » très subjectif et spécialement conçu pour un type particulier de tâches de modernisation. On pouvait créer une expérience pour la modernisation d'un mainframe.

On pouvait en créer une autre pour les mises à niveau Java, et encore une autre pour Angular React, car il fallait compenser les lacunes des capacités du modèle par ce genre de « superstructure » externe, si l'on peut dire. Ce n'est plus vraiment nécessaire aujourd'hui.

Les modèles sont désormais suffisamment performants pour qu'il soit préférable, à mon sens, de les traiter de manière abstraite et de supprimer ces structures de soutien. J'ai constaté qu'essayer d'utiliser les structures de soutien que j'utilisais il y a six mois conduit aujourd'hui à de moins bons résultats, car elles limitent et entravent la capacité de l'agent à résoudre les problèmes de manière créative et à trouver des solutions aux difficultés qu'il rencontre.

Vous allez donc probablement m'entendre répéter ce refrain. Vous pouvez simplement passer à l'action maintenant.

Les modèles sont tout simplement capables de faire des choses. Ne laissez pas vos idées préconçues d'il y a quelques mois influencer la façon dont vous envisagez de faire les choses aujourd'hui.

J’ai utilisé Fable pour de nombreuses migrations de code, et ce n’est même plus vraiment nécessaire. Je pense qu’Opus 5.

5 me donne à peu près les mêmes résultats que lorsque je fais ce genre de choses, pour la moitié du coût, voire moins. Le coût est donc en train de baisser.

La barrière à l’entrée diminue. Du genre, on peut tout simplement se lancer maintenant.

C’est vraiment passionnant. Revenons à la démo.

Désolé. Nous avons donc ici une architecture et un flux de données qui ont été extraits.

Et ensuite, il commence à extraire les domaines métier et à dire, par exemple : « D’accord. Je pense que ces fichiers contribuent à ces fonctionnalités. »

Encore une fois, tout cela est extrait par l’agent. Et il va commencer à se faire une idée de ce qu’est ce logiciel. Que fait-il quand je vais le réimplémenter ? De quoi dois-je m’assurer qu’il dispose ? Et il a également identifié une certaine dette technique ici.

Il y a des chemins d’accès en texte clair, des mots de passe en texte clair. Il y a une fonctionnalité d’importation-exportation défectueuse.

Il y a beaucoup d’éléments qui peuvent être intentionnels ou non, et nous allons donner la possibilité à des humains d’intervenir et de dire à l’agent : « Hé, cette situation d’arrondi, en fait, c’est intentionnel. »

Veuillez ne pas modifier cela, car une transformation aveugle ne vous permettrait pas nécessairement d'intégrer ce contexte métier qui n'apparaît pas dans le code. Et, bien sûr, nous avons malheureusement constaté quelques failles de sécurité.

Je pense que, plus tard, vous verrez qu’il a également détecté une violation de la norme PCI, ce qui, pour les clients du secteur des services financiers, j’imagine, pose un petit problème. Nous allons donc corriger tout cela.

N'est-ce pas ? Très bien. À présent, l'outil utilise le graphe de dépendances qu'il a établi précédemment pour créer une carte de cette base de code, et je pense que c'est l'une de mes parties préférées.

Vous pouvez voir ici, dans la barre latérale, que chacune de ces petites cases représente un fragment de code COBOL de taille représentative dans le logiciel. Et au fur et à mesure que nous avançons, pendant qu’il lit les fichiers, vous allez

pouvoir voir exactement ce qu'il examine, ce sur quoi il travaille, ce qu'il transforme, et quelle proportion du code total a déjà été transformée.

Et cela a permis de créer notre topologie. Encore une fois, c'est quelque chose que le plug-in fait tout seul.

Vous pouvez l'exécuter dès maintenant sur votre propre code si vous le souhaitez. Et nous disposons de cette superbe carte interactive de l'ensemble de la base de code de tout ce COBOL dans l'application de démonstration « Card ».

Nous avons les bases de données. Je peux cliquer sur chacune d'entre elles.

Le système a déduit la capacité, la fonction de chaque élément ici présent et ce qu'il fait. Je peux suivre chaque connexion entre n'importe quelle entité ou n'importe quel programme ici.

Et ce que je trouve vraiment génial, ce sont les flux métier et les user stories extraits. Par exemple, la mise à jour nocturne lorsque les achats d'un titulaire de carte sont enregistrés sur son compte : comment cela se passe-t-il en coulisses ? Comme ça.

On voit que ça commence à l'étape 1, je ne sais pas exactement où. Quelque part par ici.

Il y a au moins une étape n° 2. Et on peut suivre le flux exact des appels de fonction et des données pour mieux comprendre le fonctionnement.

Même si vous n'êtes pas encore prêt à moderniser votre code, le simple fait d'avoir cette vue d'ensemble et de vous dire « ah, c'est comme ça que ça marche » peut s'avérer utile. Ça peut vous aider à démêler dans votre esprit ce sur quoi vous travaillez.

Tu sais, si tu as hérité d'un gros enchevêtrement de code écrit il y a trente ans et que plus personne ne comprend vraiment, c'est le genre de chose qui te permet de comprendre, et ça va t'aider à aller de l'avant. J'adore ça.

Je trouve ça vraiment astucieux. Passons à autre chose : nous avons des diagrammes d'architecture plus détaillés générés par l'agent.

Celui-ci s'est avéré un peu enchevêtré. C'est une lacune de Mermaid.

On pourrait probablement demander à l'agent de faire un peu mieux ici. Mais, vous voyez, on peut en fait visualiser les appels de fonction d'une manière différente ici.

On voit les chemins pour cette publication nocturne, ainsi que pour d'autres fonctionnalités que cette application COBOL exécute. On commence donc à comprendre en quelque sorte la structure de ce code : quels fichiers communiquent entre eux, où les données sont stockées, à quoi ressemblent les relations entre les entités.

On commence en quelque sorte à se faire une image globale, comme une carte de la Terre qu'on est en train de regarder ici. Mais cela ne suffira pas pour transformer réellement le code et s'assurer qu'il fait ce qu'on veut en arrière-plan.

Pour y parvenir, nous devons avoir une certaine compréhension des règles métier et des fonctionnalités mises en œuvre par ce logiciel qui comptent réellement pour un utilisateur au quotidien. J'y ai déjà fait allusion brièvement.

En général, les règles métier qui comptent réellement pour l'utilisateur final d'un logiciel ne peuvent pas toujours être extraites du code avec une certitude et une précision de 100 %. Même si vous disposez d'une équipe composée des ingénieurs logiciels les plus chevronnés au monde ou de modèles de niveau « Fable plus plus plus plus » travaillant sur le projet, il existe parfois un contexte fondamental qui n'apparaît pas dans le code et que vous souhaiterez probablement obtenir auprès des personnes qui, en fin de compte, utiliseront ce code.

L'agent va donc effectuer une première analyse pour identifier et extraire les règles métier du code. Comme vous pouvez le constater, il a extrait 323 règles : 41 concernent les calculs, 146 la validation des cartes, ainsi que des éléments relatifs au cycle de vie et aux politiques.

Il a trouvé de nombreuses règles et les a classées en fonction de son niveau de confiance. Certaines sont donc tout simplement évidentes et d'une simplicité enfantine, et l'agent dira : « Ne perdons pas de temps à essayer de vérifier cela. »

C'est assez simple. Votre temps est limité.

Mon temps est limité. Le temps de chacun est limité.

Alors confions tout ce que nous pouvons aux agents et essayons d'utiliser notre temps humain limité de la manière la plus efficace possible pour clarifier les éléments que l'agent estime les plus importants pour la réussite de cette transformation. Nous avons donc ici tout un ensemble de règles métier qui ont été extraites.

Elles ont été classées par ordre de priorité en fonction de l'importance qu'il y a à ce qu'un humain les examine. On peut cliquer sur une règle métier, et elle s'affichera en langage clair.

Après chaque autorisation, pour une carte connue, le récapitulatif des comptes en attente d'autorisation est actualisé, ainsi que les limites de crédit et de retrait des comptes, etc. Et le système me demandera : « Est-ce correct ? Voici un défaut suspect.

Est-ce un comportement prévu ? Est-ce bien ce que vous souhaitiez faire ? Et le degré d'implication que vous accordez à ces règles dépend vraiment de vous. Cela dépend de votre cas d'utilisation et de la gravité du problème sur lequel vous travaillez.

Vous pouvez faire appel à des humains pour qu'ils vérifient tout cela manuellement, un par un, et s'assurent que tout est correct. Vous pouvez dire à l'agent : « Je ne sais pas.

Fais de ton mieux. Tu sais, ces deux approches sont tout à fait valables.

Tout dépend vraiment de votre tolérance au risque et du degré de précision que vous exigez pour le résultat final. Mais en fin de compte, c'est l'agent qui vous propose cette approche, et c'est à vous de prendre la décision en fonction de la sensibilité de la modernisation sur laquelle vous travaillez.

Et puis nous avons également une autre vue qui vous montre concrètement : « Voici le code COBOL. Voici la règle métier extraite de ce COBOL », juste pour vérifier une troisième fois que vous extrayez bien l'intention et que le système comprend correctement ce que vous souhaitez faire.

Et vous pouvez, par exemple, transmettre ces informations à d'autres personnes de votre entreprise. Vous pourriez demander à l'agent, si vous avez configuré le MCP Slack, par exemple :

« Salut. Peux-tu transmettre la règle métier untel à Harsh, car je pense qu'il est l'expert en la matière, et t'assurer qu'il la comprenne bien ? »

Et quand Harsh répondra sur Slack, l'agent d'ici en sera informé, enregistrera les résultats sur le disque et continuera son travail. Encore une fois, il s'agit simplement de code Claude, donc c'est extensible à l'infini.

Vous pouvez communiquer avec n'importe quel autre système dont vous disposez via MCP. Vous pouvez en faire ce que vous voulez.

Tout ce plug-in est, encore une fois, open source. Vous pouvez vous l'approprier et l'adapter à votre entreprise.

Ici, je lui donne simplement mon avis sur certaines règles métier. Il va donc maintenant rédiger un rapport.

Ce rapport, c'est en gros le document que vous pouvez présenter à votre patron en lui disant : « Voilà, je pense avoir un plan pour moderniser ce… disons, ce tas de boue avec lequel on se débat depuis un certain temps.

Voici comment je vais m'y prendre. Voici par où je vais commencer.

Voici comment je vais vérifier que tout fonctionne correctement. Voici ma compréhension des règles métier et des exigences inhérentes au code de départ, et voici comment nous allons nous assurer que le résultat final correspondra bien à ce que nous souhaitons.

Encore une fois, vous pouvez simplement vous lancer sans réfléchir. J'imagine que les secteurs réglementés sont un peu réticents à cette approche de nos jours ; c'est donc ainsi que vous renforcez cette confiance supplémentaire en suivant une sorte de processus normatif qui est raisonnable.

Voici donc notre rapport. Il cite les fichiers réels présents sur le disque à partir desquels ce rapport a été établi, à des fins d'auditabilité et de traçabilité, pour permettre une révision visant à s'assurer que, pour des raisons de conformité, vous comprenez tout ce qui a été intégré dans la modernisation de votre code. Je recommande vivement de veiller à ce que l'ensemble du répertoire dans lequel l'agent travaille et les fichiers qu'il écrit soient stockés dans Git.

Vous pouvez ainsi faire en sorte que l'agent effectue un push à chaque fois qu'il accède à un fichier ou qu'il le modifie, ce qui vous offre un suivi très granulaire et précis indiquant quel fichier a été modifié, à quel moment, et quand une décision a été prise. Si quelqu'un a validé une règle métier, cela apparaîtra sous la forme d'une modification de fichier, marquée en Markdown, dans le répertoire de travail de l'agent.

Et si vous effectuez fréquemment des poussées vers Git, vous pouvez toujours déterminer exactement à quel moment un changement s'est produit et quel utilisateur l'a autorisé. Vous pouvez ainsi expliquer de manière reproductible et en toute confiance pourquoi le code final s'est présenté sous cette forme.

Et puis, juste pour clarifier un autre point, il ne s'agit pas uniquement d'exécuter des inférences. Les agents sont désormais très doués pour écrire leurs propres outils.

Et si j’en ai l’occasion, j’essaierai de donner un exemple. Par exemple, la génération du graphe de dépendances de tous les modules du code.

Pour ce faire, il ne suffit pas simplement de faire passer tout le code par un LM. Ce serait un gaspillage incroyable, très coûteux et non déterministe.

Il va écrire de petits utilitaires Python, de petits scripts Bash pour lui-même, et les stocker pour plus tard afin que vous puissiez les consulter et vous dire : « D'accord. Lorsque l'agent a construit son graphe de dépendances précédemment, voici le script Python qu'il a exécuté ; c'est un script déterministe.

Ce script s'exécutera de la même manière à chaque fois, ce qui nous donnera un niveau de confiance bien plus élevé : ce que fait l’agent n’est pas simplement quelque chose d’éthéré et de temporaire. C’est reproductible.

Nous avons donc défini un objectif pour ce projet pilote que nous allons proposer dans notre dossier. Nous avons identifié une architecture cible, et, bien sûr, vous pouvez indiquer à l’agent que vous avez un avis différent et que vous souhaitez procéder autrement, en vous appuyant sur une pile différente, en adoptant une architecture différente, etc., et il s’adaptera en conséquence.

Il proposera une correspondance entre chaque composant hérité et un nouveau composant de la nouvelle architecture. Encore une fois, dans ce cas précis, il s’agit de passer de Cobalt à Java, deux architectures totalement différentes.

Il s’agit en quelque sorte de prendre les règles métier, puis de construire une architecture cohérente à partir de zéro. Il y a ici un certain nombre de remarques à examiner par des humains, et un ordre d’exécution est proposé.

Ainsi, en particulier pour les très grands projets, vous ne voudrez probablement pas tout changer d’un seul coup, car si vous le faites et que vous découvrez un problème dans une partie du code, vous devrez refaire le reste, ce qui prend du temps et coûte de l’argent, et vos ressources seraient sans doute mieux utilisées pour des tâches plus spécialisées que cela. Dans ce cas précis, il a donc identifié une partie du code relativement peu connectée, qui ne dépend pas énormément du reste du code, pour la séparer en premier lieu du COBOL d’origine à l’aide de la technique du « strangler fig », puis, à l’aide de quelques shims, la faire fonctionner avec le COBOL d’origine tout en s’assurant que le comportement reste sensiblement le même.

Le projet pilote va donc transformer cette unité représentative, et il va nous expliquer comment il recommande de mener ces phases, simplement en se basant sur sa décomposition du code sur lequel nous travaillons ici. Et pour être

très précis, il va proposer le cycle de mise en ligne et d'introduction nocturne comme module par lequel nous commencerons.

Sans plus attendre, transformons donc un peu de code. Je vais approuver le plan.

Je vais lui demander de modifier le plan, puis je l'approuverai. Très bien.

Il a défini des tests et des règles qui, selon lui, doivent être respectés pour que ce projet pilote soit considéré comme réussi et que l'on passe à la phase suivante. Mettons-le en œuvre.

Lecture du cahier des charges, vérification de la chaîne d'outils. Il va compiler le code d'origine et le code résultant, puis effectuer des tests pour s'assurer que les comportements sont identiques.

Il rédige quelques tests supplémentaires, puis procède enfin à la mise en œuvre, ce qui, honnêtement, est la partie la plus rapide de tout ce processus. Il comporte une analyse de l'architecture.

Il examine le code résultant pour s'assurer qu'aucun anti-modèle n'a été utilisé, et vous pouvez lui fournir un contexte supplémentaire ainsi que les éléments que vous souhaitiez éviter. Dans ce cas précis, comme nous passons à Java, qui s'expose comme un serveur web classique, il a été identifié que l'implémentation proposée comporte un déclencheur REST ouvert, permettant à un utilisateur non authentifié de lancer une opération de transfert d'argent.

Très mauvais. Vraiment très mauvais.

On ne veut pas autoriser cela, et l'agent l'a détecté pour moi ici. On peut donc voir le module en cours de transformation ici, et c'est terminé.

La transformation a eu lieu, et l'équivalence a été vérifiée. Ce que vous pouvez voir ici, ce sont les 18 règles qui concernent ce module particulier, le module de calcul des intérêts dont nous avons parlé.

Il me montre très précisément : voici le code COBOL qui implémente une règle particulière, dans ce cas précis, la lecture des soldes par catégorie dans l'ordre des clés. Vous pouvez également voir, par exemple, que l'intérêt correspond au produit du solde par le taux, divisé par 1 200, puis tronqué.

C'est donc là que ce calcul s'effectue réellement dans le code. Il s'agit de la règle que l'agent a extraite à partir de sa compréhension de ce qu'il observe dans le code.

Et voici le code Java qui implémente la même fonctionnalité. Vous pouvez donc très facilement voir, par exemple, « avant » et « après », ce que l'agent pense qu'il se passe, ainsi que les tests qui vérifient ce comportement pour s'assurer que le comportement « avant » et « après » est identique, et vous pouvez lui demander de générer cela pour l'intégralité du code si vous le souhaitez.

Ce n'est pas si compliqué. L'agent s'en sort très bien et cela permet de s'assurer qu'il fait effectivement ce qu'il faut.

Et tout cela n'est que du code HTML généré dynamiquement. J'ai simplement donné une petite indication à l'agent pour lui dire : « Hé.

Je veux voir une comparaison côte à côte. Je crois que c'est déjà intégré dans le plug-in, donc tu ne devrais même pas avoir à le faire.

Mais, vous voyez, ils sont tellement flexibles maintenant que si vous dites : « Je veux ça, mais je veux qu'il y ait des licornes et des arcs-en-ciel dessus », ou « Je veux ça, mais je veux, vous savez, une quatrième colonne qui affiche autre chose », vous pouvez tout simplement le faire.

On n'est plus limité par ce qu'une application de modernisation rigide et spécialisée peut offrir. On peut simplement l'afficher comme on le souhaite.

Voici donc notre preuve d'équivalence. À partir d'une même entrée, les deux programmes – l'application COBOL héritée que j'ai exécutée localement avec le nouveau COBOL et la version Java – ont été soumis à une série de cas de test, à savoir un ensemble de transactions simulées via COBOL, et ont affiché un comportement identique.

Il a détecté un code de sortie anormal pour certaines de ces applications, ce qui est correct, c'est-à-dire le comportement souhaité, et il a identifié un cas de défaillance potentielle qu'il a évité, et il souhaite vérifier que c'est bien la bonne chose à faire. Dans ce cas précis, au lieu d'arrondir à l'unité supérieure ou de tronquer, on obtiendrait un résultat différent.

Cela se traduirait par un nombre différent de centimes sur les comptes des utilisateurs en fin de journée. Une transformation naïve pourrait tomber dans ce piège, et vous vous retrouveriez avec, disons, quelques centimes d'écart, votre comptabilité ne fonctionnerait plus et tout le monde serait très en colère contre vous, ce qui est une mauvaise chose.

C'est donc le genre de détail qui est intégré dans le plug-in et dans l'agent pour être systématiquement recherché et pris en compte. Et maintenant, vous pouvez

voir ici qu'après ce petit exercice amusant que nous avons fait, nous en sommes à 2 % de modernisation de ce code, ce qui peut sembler peu.

N'est-ce pas ? Mais cela nous a offert de nombreuses occasions de nous assurer que notre compréhension du fonctionnement prévu du code existant est correcte. Nous avons la possibilité d'impliquer des parties prenantes d'autres services de l'entreprise afin de vérifier que le comportement final correspond bien à ce qu'elles souhaitent voir et à ce qui leur permettra de réussir.

C'est là tout l'intérêt de ce workflow, l'intérêt de doter des agents de plus en plus intelligents d'un comportement structuré. Il s'agit délibérément d'un module complémentaire, et non d'un produit conçu sur mesure, car je m'attends à ce que, à mesure que les modèles gagneront en capacités, une partie de plus en plus importante de cette structure de soutien disparaisse et devienne superflue, et nous l'adapterons au fur et à mesure.

Mais je pense qu'il y aura toujours un besoin de comprendre ce contexte métier propre à l'entreprise, propre aux personnes qui vont utiliser votre logiciel, et cet outil est destiné à vous fournir de nombreux points d'ancrage pertinents, qui permettent d'optimiser le temps consacré par les humains à intégrer ce contexte, cette connaissance métier et cette expertise du domaine dans votre code avant de produire quelque chose et de le diffuser au monde entier pour qu'il soit testé. Nous voyons donc ici que nous avons réalisé une cartographie complète de l'application de démonstration sur l'automobile, que nous avons transformé un module, et j'imagine que vous avez une assez bonne idée de la manière dont nous allons procéder pour le reste.

Nous avons donc ici notre topologie finale. Elle ressemble à ce qu'elle était auparavant, mais si vous regardez ce petit élément-là, vous pouvez voir qu'il a été réécrit en Java.

Nous avons 148 tests réussis qui prouvent qu'il fonctionne correctement. Et, au fur et à mesure que nous avançons dans la transformation, nous pourrons voir comment l'architecture évolue dans cette vue-ci.

Il y a une autre chose que je trouve amusant de vous montrer à tous : Opus 5. 5. C'est en quelque sorte un sujet secondaire dans le cadre de la modernisation, mais je vais quand même vous le montrer tant que j'ai un public captif. Cet outil est désormais très performant pour créer des scènes en 3D.

Je lui ai donc fourni un MCP vers Blender et je lui ai demandé, en une seule instruction, de me donner un rendu génial d'un mainframe, en train de se

moderniser et de s’envoler vers le cloud, et ça a été fait en une seule instruction. On peut désormais faire des choses amusantes.

C’est très passionnant. Je pense qu’il nous reste un peu de temps pour que je vous présente très rapidement une application Java bien plus simple.

Et quand je dis « bien plus simple », on parle d’une application Java d’un demi-million de lignes de code. Je vais donc la parcourir très rapidement.

La partie importante que je veux vous montrer se trouve à la fin, lorsque nous arrivons à la validation. Et si vous êtes très, très perspicaces, vous remarquerez qu’au fur et à mesure que le processus avance, il va suivre différentes étapes, mais en moins grand nombre que pour la modernisation du mainframe. En effet, il n’est pas nécessaire de procéder à toute cette extraction des règles métier si vous vous contentez, par exemple, de faire passer Java de la version 8 à la version 17, ce qui est le cas ici.

On peut raccourcir considérablement le processus. En gros, on peut simplement effectuer la mise à niveau, voir ce qui ne fonctionne plus, lancer une série de tests, essayer de déboguer soi-même les éléments défaillants, puis signaler aux humains les problèmes vraiment douteux.

Mais pour les opérations plus simples, encore une fois, le passage d’Angular à React, Python Node, les mises à niveau du runtime Java, etc., le système peut en quelque sorte s’en charger tout seul à ce stade. Voici donc ce que je voulais vous montrer.

Voici l'ensemble des tests de l'application Java d'un demi-million de lignes de code que j'ai utilisée. J'ai utilisé Jetty, qui est en quelque sorte un très ancien serveur web Java, développé sous Java 8 dans les années 90, et je suis en train de le mettre à niveau vers Java 17.

Chaque case verte que vous voyez correspond à un test qui vient de fonctionner correctement, un test qui faisait déjà partie de la suite de tests de Jetty, après notre mise à niveau. Et lorsque nous l’avons mise à niveau, cela a mis à jour Palm XML, Maven et d’autres composants, puis le système a recherché les éléments qui ne fonctionnaient plus, s’est auto-débogué… En fait, ce n’est pas très différent de la façon dont on procède pour mettre à niveau du code agentique en Java, disons, il y a six mois.

Mais ce qui est génial, c’est que l’agent est très efficace pour expliquer ce qu’il fait et où il a besoin d’aide. Vous pouvez voir ici qu’il y a 23 différences.

Je peux examiner chacune d'entre elles en détail pour comprendre ce qui a changé, pourquoi c'est important, si ça m'intéresse, s'il s'agit d'un comportement intentionnel ou non, et de vraiment concentrer mon temps sur les véritables petits détails et les petites différences de comportement entre Java 8 et Java 17 qui comptent pour moi au final, plutôt que de m'attarder sur toutes ces choses simples et futiles qui fonctionnent tout simplement si j'augmente le numéro de version dans le fichier XML du POM. Merci donc à tous de m'avoir écouté.

Je suis vraiment enthousiasmé par tout ce qu'on peut faire aujourd'hui dans le domaine de la modernisation du code. Les modèles ne cessent de s'améliorer.

On peut tout simplement faire des choses. C'est incroyable.

Je vous recommande vraiment à tous de vous amuser un peu avec ça. Et ces problèmes que vous pensiez insurmontables il y a six mois, trois mois, voire deux mois…

Les frontières ne cessent d'évoluer, et ce que vous pouvez faire aujourd'hui est différent de ce que vous pouviez faire hier. Alors amusez-vous, expérimentez, lancez de petits projets pilotes, essayez des choses.

On vit en ce moment dans un futur incroyable où les logiciels sont super amusants et où, pour les tâches ennuyeuses, on peut simplement demander à Claude de s'en charger, ce qui est vraiment génial. Je tiens à m'assurer de laisser du temps pour les questions, je vais donc redonner la parole à nos charmantes modératrices, si je trouve le moment opportun pour le faire.

Belinda Neal

44:59 - 45:19

Excellent. Merci, Morgan.

C'était un tour d'horizon vraiment passionnant, et j'adore la façon dont tu as présenté les choses. On peut tout simplement faire des choses maintenant.

Nous avons donc beaucoup de questions ici, que nous allons aborder, et merci beaucoup à tous de les avoir envoyées. Morgan, pourquoi ne pas y jeter un œil, et nous verrons par où commencer.

Morgan Lunt
45:19 - 47:13

Oui, très bien.

Je vais commencer par le haut. Quel est le plug-in que tu as utilisé dans ton code Claude pour afficher la scène sur le côté droit ? Oui.

Il s'agit du plugin de modernisation du code, dont je crois que nous avons partagé les liens, et que j'ai moi-même développé. Encore une fois, je ne dis pas qu'il est parfait, mais c'est un point de départ.

N'hésitez pas à l'adapter à votre guise. J'ai amélioré ce plugin il y a quelques jours pour tirer parti d'une fonctionnalité appelée « Anthropic mods », qui sera lancée très prochainement.

Vous avez donc tous eu un petit aperçu d'une version préliminaire. Je vais publier une mise à jour du plug-in dans les prochains jours qui inclura cette barre latérale avec ce mod, et c'est justement ce mod qui permet la personnalisation effective de l'expérience utilisateur.

Bref, jetez-y un œil en milieu de semaine prochaine : le plugin disponible sur GitHub inclura cette barre latérale. Il fonctionnera tout simplement pour vous.

On dirait qu'on en a une autre. Comment vous assurez-vous qu'aucune fonctionnalité ne soit perdue lors de la migration, et quelles mesures de sécurité mettez-vous en place pour éviter les régressions ? C'est en quelque sorte un mélange de tests existants complétés par de nouveaux tests.

Vous pouvez demander à Anthropic de rédiger de nouveaux tests, ou opter pour des déploiements bleu-vert. Le type de validation dépendra vraiment de votre état initial, de votre état final, et du niveau de précision que vous souhaitez atteindre dans vos garanties de parité.

J'ai vu des équipes faire toutes sortes de choses, comme faire tourner des bases de données en parallèle et effectuer des déploiements « bleu-vert » d'applications modernisées pendant des semaines, voire des mois, afin d'identifier tout écart dès son apparition, d'en analyser en profondeur la cause et de modifier le code pour y remédier. Je suis, disons, extrêmement prudent.

Et si vous êtes un peu moins réfractaire au risque, alors peut-être que tous les tests déjà intégrés à votre code suffisent. Et s'ils présentent les mêmes comportements dans votre suite de tests, vous pouvez y aller.

C’est en quelque sorte une décision à prendre au cas par cas, en fonction de votre tolérance au risque pour le logiciel sur lequel vous travaillez.

Harsh Patel
47:13 - 47:29

Morgan, un autre sujet qui revient souvent ici, c’est de savoir comment rendre ce plug-in peut-être plus personnalisable. N’est-ce pas ? Beaucoup d’entreprises ont leurs propres compétences qui reflètent leurs styles de codage ou ce que vous voulez.

Comment aborderiez-vous cette question ?

Morgan Lunt
47:29 - 48:08

J'installerais le plug-in, puis je dirais à Claude : « Salut. Mon entreprise utilise cette compétence.

Nous écrivons le code de cette manière. Nous avons ces normes organisationnelles.

Modifie le plugin pour qu’il respecte ces normes à chaque étape du processus. Et ce qui est vraiment hallucinant avec Claude Code, c’est qu’il peut écrire des mods, des plugins, des compétences ou des modifications pour modifier Claude Code lui-même.

Du coup, c’est simple : identifiez quelque chose dont vous voulez changer le comportement, dites-lui simplement de se comporter différemment, et testez le résultat.

Et si ça ne fait pas ce que vous voulez, orientez-le dans une autre direction et continuez à l’orienter jusqu’à ce qu’il fasse ce que vous voulez.

Harsh Patel
48:08 - 48:25

L’une des meilleures pratiques que j’ai observées ici, c’est que l’on peut récupérer toute la documentation. Celle-ci se présente généralement sous la forme d’un fichier .txt, et il suffit de l’utiliser comme données d’entrée pour que Claude puisse faire exactement ce que vous venez de mentionner.

Morgan Lunt
48:25 - 49:37

Oui. Est-ce que le plug-in peut être utilisé pour n'importe quelle langue source ? Encore une fois, je m'en remets entièrement à nos équipes de recherche, mais les modèles sont suffisamment performants pour que je n'aie pas encore rencontré de langue qu'ils ne puissent raisonnablement gérer et traiter.

Par exemple, je pensais que le COBOL allait leur poser des problèmes. Pas vraiment.

J’ai vu des langages étranges et ésotériques dont je n’avais jamais entendu parler auparavant être gérés relativement bien. Si vous identifiez un langage avec lequel Claude Code ne fonctionne pas bien, j’aimerais le savoir.

Je serais en effet très curieux, et nous pourrions probablement orienter la recherche pour créer des environnements et organiser des formations à ce sujet afin d’améliorer l’ensemble du processus. Comment surveillez-vous le coût d’une migration avec le plug-in de modernisation ? La méthode la plus simple consiste simplement à créer une clé API pour ce projet de modernisation spécifique, puis vous pouvez simplement identifier, par exemple : « Tiens.

Toute utilisation liée à ce projet passe par cette clé API. Vous pouvez ainsi suivre l’évolution de la facture.

Pour donner un ordre de grandeur (ne me prenez pas au mot), cette mise à niveau de 500 000 lignes de code Java que j’ai montrée a, je crois, coûté au final quelques centaines de dollars en jetons. Rien d’exorbitant.

Harsh Patel
49:37 - 49:55

Une autre question qui revient assez souvent : vous expliquez comment, au fur et à mesure que nous avançons, le système est capable d'extraire les règles métier. Comment vous assurez-vous de récupérer toutes les règles métier, comme vous l'avez montré lors de cette analyse contradictoire ? Pouvez-vous nous en dire un peu plus sur les workflows dynamiques qui sous-tendent en grande partie ce processus ?

Morgan Lunt
49:55 - 51:17

Oui. Merci, merci d'avoir soulevé ce point.

L'une des choses qui permet à Anthropic de raisonner sur des modernisations à grande échelle comme celle-ci, c'est qu'il y a des dizaines, voire des centaines de millions de tokens dans cette base de code. N'est-ce pas ? Et nos fenêtres de contexte restent limitées à environ un million sur tous nos modèles.

On ne peut donc pas se contenter de charger naïvement l'intégralité du code dans le modèle. À la place, nous utilisons une orchestration multi-agents qui s'appuie largement sur des workflows, notamment des workflows dynamiques.

Et les workflows dynamiques ne sont qu'une fonctionnalité du code de Claude : Claude écrit un script, essentiellement un script déterministe qui lance de nombreux autres Claudes, lesquels peuvent à leur tour en lancer beaucoup d'autres si nécessaire, afin de se déployer en éventail et d'effectuer très rapidement un travail parallèle à grande échelle, un peu comme une fonction MapReduce, puis d'exécuter le travail hautement parallélisé, de synthétiser les résultats en amont et de renvoyer le résumé à l'agent principal avec lequel vous interagissez réellement. Et ces agents parallèles, eux-mêmes, continuent de créer des artefacts durables.

Ils continuent d'écrire des fichiers sur le disque afin que, si leur travail doit être consulté ultérieurement, cela soit possible. Mais du point de vue de l'utilisateur, celui-ci interagit avec un seul agent principal.

Il n'y a pas vraiment de problèmes de contexte, car cet agent est capable de déléguer, via des workflows dynamiques, des sous-agents et des arborescences de sous-agents, pour effectuer le travail routinier, comme examiner un fichier JCL et déterminer quelles sont ses dépendances et ce qu'il fait.

Harsh Patel
51:17 - 51:44

Parfait. Je pense que la question suivante était très similaire à la première réponse que vous avez donnée, qui portait sur la vérification.

N'est-ce pas ? Alors, comment s'assurer que ce code généré par l'IA fonctionne correctement ? Si je comprends bien, Morgan, le plug-in ne se contente pas de vous guider, par exemple, dans la mise en place d'un harnais d'équivalence, mais nous pouvons également utiliser d'autres données qui pourraient être disponibles. Ainsi, si nous migrons un pipeline de données, nous pouvons utiliser des ensembles de données historiques pour nous assurer que les entrées et les sorties correspondent bien une à une.

Morgan Lunt
51:44 - 51:57

Oui, exactement.

C'est la même méthode que celle que vous utiliseriez pour vérifier le travail d'un consultant humain que vous auriez engagé pour mener un projet de modernisation. Il s'agit de faire fonctionner les deux systèmes en parallèle, de rechercher les écarts et les signaux, puis d'itérer.

Harsh Patel
51:57 - 54:02

Parfait. Très bien.

Je pense qu'on a reçu beaucoup de questions du même genre. On pourrait donc peut-être commencer par conclure en abordant les prochaines étapes.

N'est-ce pas ? Alors, par où commencer, et quelles sont les ressources supplémentaires auxquelles vous pourriez avoir recours ? Il y a donc beaucoup de choses différentes que vous pouvez faire. La première, bien sûr, c'est que ce plug-in est disponible en open source.

N'hésitez pas à taper cette commande : « /plug-in », puis installez le plug-in de modernisation du code. Il est mis à votre disposition.

Vous pouvez ensuite l'adapter aux besoins spécifiques de votre organisation. Encore une fois, si vous souhaitez l'enrichir davantage, vous disposez de toute la flexibilité nécessaire pour le faire, et il est indépendant du langage de programmation.

Une ressource supplémentaire, également disponible dans notre documentation, est notre guide pratique. Celui-ci est issu de l'expérience concrète acquise sur le terrain par nos ingénieurs déployés sur le terrain, qui ont travaillé avec des clients comme vous et ont appris à mener ce type d'exercices.

N'hésitez donc pas à y jeter un œil : il présente, dans un esprit très similaire, ces séries d'étapes que vous pouvez suivre pour mettre en place les contrats d'équivalence, vous assurer que le code se comporte réellement comme vous le souhaitez, ainsi que pour configurer ces différents types de « harnais », comme le harnais de code que vous intégrez, afin de pouvoir effectuer ce processus de bout en bout. De plus, bon nombre de nos ingénieurs ont également travaillé en interne sur de nombreux projets de migration de code.

N'hésitez donc pas à consulter l'un de ces exemples, qui consistait à migrer un million de lignes de code Zig vers Rust, ce que nous avons réussi à faire en moins de deux semaines. Je terminerai peut-être par un petit aperçu.

Tu sais, Morgan, tu as travaillé sur cet exemple vraiment passionnant de migration d'une base de code COBOL vers Java. Mais l'une des questions qui se posent est la suivante : à l'avenir, et si nous n'avions plus besoin de moderniser le code lui-même ? Veux-tu peut-être en parler un peu plus en termes de vision d'avenir et de la manière dont nous pouvons commencer à y réfléchir, maintenant que ces initiatives sont déjà en cours ?

Morgan Lunt
54:02 - 55:26

Oui. Dans le cadre de mon travail auprès de clients professionnels, je passe beaucoup de temps à réfléchir au cycle de vie du développement logiciel (SDLC) dans son ensemble et à la façon dont il évolue.

Avant, le goulot d'étranglement, c'était l'écriture du code, mais ce n'est plus vraiment le cas aujourd'hui. Je considère que le problème de l'écriture du code

est en grande partie résolu.

Et les points où les gens rencontrent aujourd'hui des goulots d'étranglement sont plutôt liés à la validation. Ils sont liés à la révision du code.

Elles concernent les tests d'intégration, et nous avons quelques projets en cours qui, je l'espère, aideront un peu les gens à cet égard. Ma vision est la suivante : une fois que nous aurons un peu facilité ce parcours et que nous aurons rendu le processus — non seulement l'écriture du code, mais aussi sa vérification, sa validation et son déploiement — rapide, bien rodé et ne nécessitant pas nécessairement une énorme quantité d'efforts humains tout au long de ce processus, nous pourrons commencer à faire des choses vraiment géniales, comme maintenir votre code toujours à jour.

S'il y a une nouvelle version de Python, pourquoi votre package ne se mettrait-il pas simplement à jour automatiquement, ne s'assurerait-il pas que les tests de régression sont réussis, puis ne se déploierait-il pas ? J'imagine un monde, d'ici peu, où la modernisation ne fera plus vraiment l'objet de discussions. Ce ne sera plus une tâche que les gens devront accomplir.

Une fois que vous aurez atteint cet état, vous pourrez garantir que vous resterez toujours à la pointe en apportant des modifications incrémentielles chaque semaine, chaque mois, à votre code, et en devançant tout grand projet de modernisation qui résulterait d'un code resté statique pendant un certain temps avant de commencer à s'atrophier.

Harsh Patel
55:26 - 55:41

Oui. Ça me parle vraiment.

Chaque fois que nous discutons avec des clients, beaucoup d'entre eux ont ces cycles trimestriels où ils doivent se livrer à ce genre d'exercices. Donc, être capable de faire cela va complètement changer notre façon de concevoir les logiciels dans leur ensemble.

Belinda Neal
55:41 - 55:58

C'était formidable, Morgan et Harsh. Merci beaucoup.

Une dernière chose : dans l'onglet « Documents » du webinaire, vous trouverez tous les liens. N'hésitez pas à les consulter.

Et n’oubliez pas de nous faire part de vos commentaires après le webinaire. Nous attendons avec impatience vos questions et vos remarques.

Merci à tous d’avoir été avec nous aujourd’hui, et nous espérons vous revoir très bientôt.

Harsh Patel
55:58 - 55:59

Merci.

Morgan Lunt
55:59 - 56:00

Merci à tous.