

Ginux : construire un noyau indépendant capable d'exécuter l'espace utilisateur Linux grâce à l'ingénierie agentique dirigée par l'humain

Gregory M. Kurtzer

Rapport technique — octobre 2026

DOI 10.5281/zenodo.23682548

Source : <https://github.com/gmkurtzer/ginux>

Résumé

Cet article présente une étude de cas en ingénierie agentique dirigée par l'humain (Human-Directed Agentic Engineering, HDAE) : une architecture, des contraintes, des jalons et des critères d'acceptation définis par un humain sont associés à une boucle autonome d'implémentation, d'exécution, de diagnostic et de révision. En onze jours, une tâche persistante d'OpenAI Codex utilisant GPT-5.6 Sol a produit Ginux, un noyau de recherche x86-64 privilégiant Rust et contenant environ 139 000 lignes conservées de code d'implémentation et de test.

Ginux démarre via UEFI et fournit sa propre gestion mémoire, son ordonnanceur, son modèle de processus, ses systèmes de fichiers, son réseau, ses périphériques et une couche de compatibilité Linux. Son contrôleur s'exécute en anneau 3. Le système évalué exécute certains programmes Alpine non modifiés et termine un démarrage de Rocky Linux 10 avec systemd jusqu'à multi-user.target. Il exécute également le même binaire sysbench qu'un système Rocky Linux standard. Des itérations exploratoires guidées par des benchmarks ont ramené de graves défauts des chemins mémoire et threads à une quasi-parité sur trois charges étroites à un seul vCPU, sous traduction QEMU inter-architecture. Ces mesures sont diagnostiques et ne constituent pas des résultats généraux de performance du noyau. Le système explore aussi les domaines natifs du noyau comme unités d'isolation et d'ordonnancement, bien que ces domaines soient un résultat du projet plutôt que son objectif initial.

Le résultat démontre une intégration rapide de mécanismes systèmes éprouvés, et non une alternative à Linux prête pour la production. La compatibilité reste partielle, l'expérience ne comporte ni groupe témoin apparié ni reproduction indépendante, et la couverture matérielle, de sécurité et applicative est limitée. La conclusion circonscrite est que la HDAE peut produire des logiciels systèmes profonds, mais que la connaissance du domaine, la validation exécutable, les normes partagées et un écosystème durable restent plus importants que la seule vitesse de génération de code.

1 Introduction

La HDAE peut-elle guider un agent de programmation dans la construction d'un nouveau noyau capable d'exécuter certaines charges de travail existantes de l'espace utilisateur Linux ? Les mécanismes des noyaux sont largement documentés, mais leur intégration dans un système cohérent est une tâche de longue haleine : architecture, implémentation, diagnostic, compatibilité et évaluation doivent rester alignés au fil de milliers de changements. Ginix sert à étudier trois questions :

RQ1 — Capacité. La HDAE peut-elle produire un noyau indépendant qui démarre et exécute des charges de travail non modifiées issues de distributions Linux ?

RQ2 — Méthode. Quelle documentation, quelle orientation, quels contrôles exécutables et quelles boucles de rétroaction sont nécessaires pour soutenir le travail ?

RQ3 — Qualité du résultat. Quelle compatibilité et quelles performances le système obtenu démontre-t-il, et où s'arrêtent les affirmations possibles ?

L'article apporte le système obtenu, la méthode HDAE documentée et un compte rendu fondé sur des preuves de ses capacités et limites. Ginix est un noyau indépendant ; Linux constitue sa cible de compatibilité pour l'espace utilisateur. Un domaine doté d'une personnalité Linux présente Ginix comme son noyau. Dans cet article, « exécuter l'espace utilisateur Linux » signifie uniquement que les programmes nommés et testés observent suffisamment de leur ABI attendu pour fonctionner. Cela n'implique ni prise en charge de toute une distribution, ni conformité complète des appels système, ni compatibilité Linux intégrale.

Le nom est une plaisanterie délibérée dans la même famille que Linux. « Linux » combine Linus et Unix ; « Ginix » joue sur Linux, GPT et Greg. Par retenue de l'auteur, le « G » est considéré comme GPT. L'expérience délimite tout aussi délibérément son périmètre : elle étudie la capacité de l'IA à construire un noyau de recherche ciblé, et non la possibilité pour onze jours de code généré de rivaliser avec des décennies d'ingénierie Linux.

Cet article évalue le commit `d11c126a048a51fe7eb9c4105f31760add1027ea`, daté du 23 octobre 2026. Le tableau 1 décrit son ampleur. Les chiffres comptabilisent le texte du dépôt ; ils ne disent rien de sa correction ni de sa valeur technique.

Mesure	Valeur
Durée de développement	11 jours calendaires
Commits	1 037
Fichiers suivis	261
Fichiers source Rust	63
LOC implémentation/tests, charge générée comprise	174 736
LOC conservées hors génération	138 790
LOC de documentation Markdown	13 074

Tableau 1 : ampleur du dépôt au commit évalué.

2 Travaux connexes

La plupart des évaluations d'agents de programmation étudient des changements circonscrits dans des dépôts existants. SWE-bench évalue la résolution de problèmes à partir des tests du dépôt [5] ; SWE-EVO étend ce modèle à l'évolution logicielle coordonnée [12]. NL2Repo-Bench part plutôt d'exigences et d'un espace de travail vide, et constate des échecs persistants de planification, de cohérence entre fichiers et d'achèvement sur de longues trajectoires [4]. Les travaux de METR sur l'horizon temporel mesurent de même l'évolution du taux de réussite selon le temps qu'un praticien expérimenté consacrerait à la tâche [6]. Ginix n'est la soumission à aucun de ces benchmarks : c'est une étude longitudinale de cas avec une cible définie par l'humain et évoluant au fil du projet.

2 Travaux connexes (suite)

L'expérience publique la plus proche portant sur un projet complet est le compilateur C de Carlini, construit par 16 agents Claude parallèles [1]. Près de 2 000 sessions et environ 20 000 USD ont produit quelque 100 000 lignes de Rust capables de compiler des logiciels substantiels, dont la majeure partie d'une compilation du noyau Linux. Les suites de tests du compilateur et les outils de référence fournissaient des oracles particulièrement solides. Linux se distingue par l'emploi d'une tâche persistante unique dans un processus HDAE et par des contrôles comportementaux qui ont dû être créés en même temps que le noyau. Les deux expériences font du harnais et des critères d'acceptation une partie du résultat, et non de simples outils auxiliaires.

Les LLM ont également été appliqués au développement des systèmes d'exploitation. CrashFixer, par exemple, évalue un agent de réparation des plantages du noyau Linux à l'échelle d'un dépôt [8]. Ce travail améliore un noyau établi ; Linux demande si un agent peut intégrer un noyau ciblé à partir de zéro. Les recommandations de la communauté Linux concernant les contributions générées soulignent que les contributeurs doivent comprendre, examiner, tester et assumer la responsabilité du résultat [7]. Les discussions du Maintainers Summit 2026 ont également considéré le prototypage et la génération de tests comme des usages prometteurs, tout en mettant l'accent sur le coût de la revue, la compréhension des contributeurs et la maintenabilité [3]. Ces préoccupations encadrent l'interprétation présentée ici : une construction rapide prouve une capacité, mais ne remplace pas la responsabilité de maintenance.

Cet article emploie l'expression « ingénierie agentique dirigée par l'humain » comme étiquette descriptive du processus à deux boucles évalué ici, et non comme revendication d'avoir inventé l'idée générale ou l'expression. Des usages publics connexes décrivent également l'ingénierie agentique comme un travail orienté par l'intention humaine, les contraintes et la revue [9, 10]. La contribution réside dans la méthode concrète, la provenance et l'étude de cas portant sur l'ensemble du système, plutôt que dans l'étiquette elle-même.

3 Méthodes

3.1 Cadre expérimental et provenance

Le travail a commencé avec une ébauche d'architecture, mais sans implémentation du noyau. L'objectif était plus précis que « construire un noyau » : bâtir un noyau indépendant avec un espace utilisateur Linux ordinaire, un hôte réduit, des frontières de privilèges fortes, des services récupérables et une expérience opérateur se comportant comme un véritable système plutôt que comme une démonstration. La période évaluée couvre onze jours calendaires. Presque toute l'implémentation a eu lieu dans une tâche Codex de longue durée ; l'auteur définissait l'architecture, exploitait le système, interprétait les résultats et prenait les décisions finales d'acceptation.

Le développement s'est déroulé sur un Mac Apple Silicon sous macOS. Linux ciblant x86-64, les compilations étaient démarrées avec `qemu-system-x86_64` et le traducteur de code TCG de QEMU, plutôt qu'avec une accélération matérielle native. La machine complète était ainsi reproductible depuis l'hôte de développement, mais les temps rapportés incluent la surcharge de l'émulation inter-architecture.

Le projet utilisait OpenAI Codex avec GPT-5.6 Sol dans l'environnement de bureau Codex. Au dernier point de contrôle documentaire, l'environnement indiquait la version de bureau 26.825.51511 (build 7377) et `codex-cli` 0.151.0-alpha.7.2. Le client étant continuellement mis à jour, ces versions identifient le dernier point de contrôle et non chaque interaction. Le choix du modèle était pragmatique, non contrôlé. Dans les travaux antérieurs de l'auteur, des modèles Claude avancés consacraient plus souvent du contexte et du budget de jetons à des pistes techniquement intéressantes mais ensuite abandonnées. Ici, Codex restait plus proche du jalon actif ; la structure de tâches et d'objectifs rendait la limite courante visible et sa boucle de travail répétait les cycles édition, compilation, démarrage, diagnostic et révision avec moins d'intervention. Aucun prompt apparié, aucune notation à l'aveugle ni mesure des jetons au niveau du fournisseur n'ont été collectés : cela explique le choix sans établir de comparaison entre modèles.

L'architecture et les plans persistaient en dehors de la conversation. L'architecture versionnée définissait le démarrage, la frontière noyau/espace utilisateur, les domaines, l'autorité, le stockage, les périphériques et les critères d'acceptation. Le registre de compatibilité consignait les comportements avec et sans appels système. Un objectif de travail, des cas d'usage et un journal des décisions maintenaient le plan et sa justification visibles. La conversation, l'historique Git, les tests et une chronologie d'orientation sélectionnée constituent le dossier de provenance.

3.2 Construction par étapes

La construction a progressé en six étapes qui se chevauchaient, chacune se terminant par un contrôle observable :

1. Démarrage et architecture : produire une image EFI x86-64 unifiée, quitter les services de démarrage, établir une pile noyau et inventorier la mémoire et l'état ACPI.
2. Mécanismes du noyau et matériel : ajouter pagination, exceptions, interruptions, ordonnancement, capacités, IPC, PCI, chemins virtio et confinement DMA.
3. Contrôle de l'espace utilisateur : entrer en anneau 3, exécuter le contrôleur statique intégré comme processus ordinaire et exposer des opérations de gestion versionnées.
4. Exécution Linux : ajouter ELF, PIE et chargement dynamique, sémantique des processus et signaux, systèmes de fichiers virtuels, terminaux, sockets et appels système.
5. Intégration des distributions : ajouter réseau et transport OCI, vérification, décompression, mise en cache et racines propres aux domaines ; exécuter ensuite Alpine et des programmes Rocky Linux de plus en plus complexes.
6. Démarrage système composé : transformer les échecs en matrice de compatibilité et exiger la persistance, le cycle de vie multi-domaines, le démarrage des services Rocky systemd et une console interactive.

L'entrée du firmware a donc précédé le contrôle matériel ; une sonde ELF limitée a précédé un processus Linux général ; les sondes ABI synthétiques ont précédé Alpine ; Alpine a précédé Rocky Linux fondé sur glibc ; et les programmes Rocky ont précédé systemd. Un message de réussite affiché ne suffisait jamais à valider une étape. Le contrôle suivant exerçait le travail précédent dans un contexte moins synthétique.

3.3 Modèle en boucle fermée de la HDAE

Chaque étape utilisait deux boucles couplées :

Boucle externe d'orientation HDAE : vision → architecture et contraintes → contrôle d'acceptation → examen humain du comportement observé → correction, repriorisation ou acceptation.

Boucle interne de l'agent HDAE : implémentation → compilation et exécution QEMU → mesure et analyse des échecs → révision → test de régression.

La HDAE associe ces deux boucles. La direction humaine fournissait l'objectif, l'architecture, la cible de compatibilité Linux, le modèle opérateur, l'interprétation des échecs et la décision d'arrêt. Codex choisissait les détails d'implémentation, modifiait le dépôt, compilait et démarrait le système, analysait les journaux, écrivait des tests, profilait les échecs et révisait le code dans les limites du contrôle actif. Il a également proposé des mécanismes tels que des capacités étiquetées par génération, la copie à l'écriture et des chemins de pages partagées préautorisés, qui restaient soumis à une revue architecturale. L'auteur humain nommé conserve la responsabilité des affirmations, de la décision de publication et du code résultant ; Codex n'est pas indiqué comme coauteur.

La boucle interne HDAE était exécutable par l'agent, sans dépendre d'une observation manuelle. Des scripts du dépôt lançaient QEMU x86-64 sans interface sur le Mac, injectaient des commandes au contrôleur, capturaient la sortie série, imposaient des délais, vérifiaient des marqueurs attendus et des conditions d'échec, puis arrêtaient l'émulateur. Codex pouvait appeler ces scripts, examiner les journaux, modifier le système et répéter le même test. Les cycles d'implémentation pouvaient ainsi se poursuivre jusqu'à la réussite d'un contrôle d'acceptation ou jusqu'à ce qu'un résultat exige une décision architecturale humaine.

3.3 Modèle en boucle fermée de la HDAE (suite)

Cette répartition du travail fait de la HDAE un modèle pratique de prototypage, de construction et de maintenance, aussi bien pour les projets nouveaux que pour les projets établis. Sur le plan opérationnel, la boucle suivait cinq étapes :

1. Les experts du domaine définissent le résultat produit, le modèle opérationnel, les objectifs exclus et les normes déterminant l'utilité du résultat.
2. Les documents d'architecture et de conception traduisent ces connaissances en contraintes persistantes, interfaces, règles de responsabilité et comportements attendus.
3. Le travail est divisé en jalons dotés de contrôles d'acceptation explicites et de points d'arrêt délimités que l'agent peut exécuter et mesurer lui-même.
4. Dans chaque périmètre, l'agent implémente, teste, diagnostique et révisé jusqu'à la réussite du contrôle ou jusqu'à ce qu'une décision doive revenir à l'examen humain.
5. L'évaluation humaine vérifie la sémantique, l'ergonomie et les effets sur l'ensemble du système ; les décisions acceptées et les nouvelles exigences découvertes sont réinjectées dans les documents, le plan et la suite de régression avant le jalon suivant.

L'utilisation pratique a modifié le plan à plusieurs reprises. Des listes de répertoires vides ont révélé des défauts de système de fichiers et de processus ; un deuxième domaine a révélé un état singleton ; systemd a mis au jour des sémantiques manquantes pour les montages, procfs, sockets, terminaux, événements et identifiants ; la latence du contrôleur a révélé des défauts d'ordonnement. Ces symptômes ont été interprétés à partir du comportement attendu de Linux et de l'opérateur, tandis que l'agent les convertissait en tests QEMU, ABI, cycle de vie et performance reproductibles.

Les travaux de performance de la section 5.2 en donnent un exemple compact. La boucle externe HDAE a rejeté les mesures non comparables, fixé le périmètre à un seul vCPU et défini le contrôle comme la réduction de l'écart sans affaiblir la correction. Codex a ensuite instrumenté les chemins critiques, modifié un mécanisme à la fois, recompilé, relancé QEMU, comparé des échantillons répétés et conservé ou annulé chaque changement. Le travail s'est arrêté lorsque les questions de recherche disposaient d'un système démontrable, de preuves documentées et de lacunes restantes explicites — et non lorsque le prototype était devenu prêt pour la production.

L'échelle modifie la taille et le nombre des contrôles, pas la boucle elle-même. Une petite modification peut tenir dans un problème circonscrit et un test de régression ; un programme important peut nécessiter de nombreux jalons architecturaux comme ceux de Linux. Un projet entièrement nouveau doit créer ses spécifications et son harnais en parallèle de l'implémentation. Un projet existant peut commencer avec ses interfaces, tests et pratiques de revue établis, puis ajouter des contrôles pour reproduire un bogue, vérifier une exigence de compatibilité, un seuil de performance, une propriété de sécurité ou une nouvelle fonctionnalité. Dans les deux cas, le produit durable n'est pas seulement le code généré, mais aussi les spécifications, les contrôles exécutables et l'historique des décisions qui encadrent les changements futurs et les sessions ultérieures de l'agent.

Un « vibe coding » sans orientation aurait été mal adapté à ce projet. De nombreux échecs produisaient une sortie convaincante tout en violant la sémantique Linux, l'isolation ou le comportement du cycle de vie. Sans connaissances suffisantes du domaine pour reconnaître ces échecs, des démonstrations plausibles auraient facilement pu être acceptées comme un travail terminé, produisant un code fonctionnel en apparence mais structurellement faible, souvent qualifié de « AI slop ». La contribution humaine essentielle n'était pas de saisir chaque étape d'implémentation, mais de définir le système, de reconnaître les comportements erronés et de fixer la prochaine limite vérifiable.

4 Conception et implémentation du système

Linux est un artefact de démarrage EFI monolithique contenant le noyau privilégié, les composants essentiels au démarrage et un contrôleur intégré en anneau 3. Il fournit des domaines de charge de travail ; OCI fournit leur contenu initial de système de fichiers, et non leur isolation à l'exécution. Cette section décrit le démarrage, les frontières, la gestion et la compatibilité Linux.

4.1 Architecture du démarrage initial

UEFI supprime plusieurs hypothèses héritées de l'ère BIOS. Le firmware charge en mémoire une application PE32+, y compris toutes ses sections, via `LoadImage()` [13]. Un système d'exploitation peut donc être livré sous la forme d'une charge utile beaucoup plus volumineuse, plutôt que de croître au moyen d'une chaîne de chargeurs d'amorçage à partir d'un secteur de démarrage de 512 octets. La qualité du firmware, la mémoire disponible, la prise en charge des systèmes de fichiers et la représentation PE/COFF imposent néanmoins des limites pratiques.

Linux exploite cette liberté avec une architecture de démarrage monolithique : une seule image `GNU.EFI` contient son noyau privilégiant Rust, l'assembleur x86-64 nécessaire, les composants essentiels au démarrage, le contrôleur intégré, le catalogue de composants et les métadonnées de récupération. Aucun système hôte de distribution, chargeur d'amorçage séparé ou `initramfs` distinct n'est nécessaire. Après avoir quitté les services de démarrage UEFI, le noyau initialise le système et mappe le contrôleur comme processus normal en anneau 3. Le contrôleur réside donc dans l'image du noyau pour le conditionnement et le cycle de publication, mais pas pour le niveau de privilège du processeur.

Ce conditionnement a simplifié une expérience qui aurait autrement dû coordonner un chargeur d'amorçage, une image noyau, un `initramfs`, un système de fichiers racine hôte et le versionnage de l'espace utilisateur. Il a également soutenu le modèle d'immuabilité du projet : le noyau, le contrôleur, l'autorité initiale et les composants requis au démarrage peuvent être mesurés comme une seule image avant exécution. Un système de production nécessiterait toutefois une stratégie mature de mise à jour, retour arrière, signature, récupération et compatibilité firmware.

4.2 Frontières du noyau et des services

L'anneau 0 contient les mécanismes qui doivent faire respecter un invariant à l'échelle du système : tables de pages et espaces d'adressage, ordonnancement, capacités, IPC, gestion des interruptions et des temporisateurs, propriété des domaines, frontière de personnalité Linux et partie de confiance du contrôle des périphériques et du DMA. L'implémentation x86-64 n'utilise pas les anneaux 1 et 2. L'anneau 3 contient le contrôleur, les charges Linux et les services natifs isolés. Intégrer un programme dans `GNU.EFI` détermine comment ses octets sont livrés ; l'anneau du processeur détermine toujours ses privilèges.

La prise en charge des périphériques est délibérément divisée. Le noyau découvre les périphériques PCI et conserve l'application des règles relatives au contrôleur d'interruptions, MSI-X, IOMMU et capacités. Le prototype comprend des chemins virtio bloc et réseau et démontre des services Rust mesurés de pilotes/protocoles dans des domaines distincts en anneau 3, avec des autorités MMIO, DMA, files et paquets limitées. Les capacités et ressources sont étiquetées avec une génération de domaine afin qu'un service en panne puisse être mis en quiescence, révoqué et remplacé sans considérer son successeur comme la même identité de sécurité. Le prototype démontre cette conception, tandis qu'un cadre général de pilotes et une matrice matérielle restent des travaux futurs.

Le volet système de fichiers est lui aussi conçu sur mesure. Le VFS du noyau et le comportement des montages présentent des racines propres à chaque domaine, assemblées à partir de couches OCI vérifiées, d'une surcouche inscriptible et de sémantiques sélectionnées de `tmpfs`, `procfs` et `devfs`. Les pages de fichiers immuables sont partagées au sein d'un domaine et copiées lorsqu'elles deviennent inscriptibles. De petites frontières en assembleur entrent et sortent d'UEFI, installent les points d'entrée des exceptions, interruptions et appels système, changent l'état de privilège et d'espace d'adressage et fournissent des sondes ABI ; les politiques et la plupart des mécanismes restent en Rust.

4.3 Contrôleur, domaines et personnalités

Le contrôleur est compilé dans `GNU.EFI` et non installé comme service hôte distinct. Au démarrage, le noyau mappe cette charge utile Rust authentifiée et liée statiquement dans l'anneau 3. Il s'agit de l'outil principal de configuration et d'administration du noyau, des périphériques et des domaines ; il utilise l'ABI Linux pour le comportement ordinaire des processus et une interface de gestion versionnée de périphériques/`ioctl`. Son interface de commande découvre l'état de l'hôte et les périphériques ; configure le réseau, le stockage, la persistance et le cache OCI ; définit les images de domaines, processus `init`, ressources, rattachements, démarrage automatique et cycle de vie ; et rapporte l'état des processus, domaines, périphériques et de préparation.

4.3 Contrôleur, domaines et personnalités (suite)

Le contrôleur appartient ainsi à l'image unique du noyau et à son unité de publication, tandis que l'administration s'exécute avec les privilèges de l'espace utilisateur. Le contrôleur ne contourne pas le noyau pour effectuer des opérations privilégiées. Il envoie des requêtes de gestion versionnées via l'interface de contrôle ; le noyau valide les capacités et la propriété avant d'effectuer l'opération protégée. La configuration est enregistrée sur le volume détenu par l'hôte pour les démarrages ultérieurs. Le contrôleur peut déléguer ou révoquer des sous-ensembles de son autorité de démarrage scellée, mais ne peut pas l'étendre.

Les interfaces physiques sont des ressources détenues par le noyau. Le contrôleur configure DHCP et DNS côté hôte, crée des liens virtuels et rattache un ou plusieurs liens à un domaine sans les renommer. Un domaine sans rattachement explicite peut partager le réseau de l'hôte, et chaque domaine reçoit une interface loopback. Le réseau hôte et le cycle de vie des liens sont testés ; les chemins de paquets et baux indépendants par domaine ne sont pas terminés.

Un domaine est un objet natif du noyau Linux dont le rôle dépasse celui d'un conteneur OCI ou d'un espace de noms Linux. Il possède ou reçoit des espaces d'adressage, un arbre de processus, des capacités, des objets IPC, des vues de systèmes de fichiers et de montages, des réseaux, des périphériques, des limites, un cycle de vie et une politique d'échec. OCI fournit le système de fichiers initial du domaine ; Linux fournit son isolation et son exécution. Un domaine unique peut ressembler à un ordinateur conventionnel, avec systemd comme PID 1 et un espace utilisateur de distribution au-dessus. Plusieurs domaines partagent un noyau tout en conservant des identités et ressources distinctes, détenues par le noyau.

Les domaines sont également des entités d'ordonnancement. L'ordonnanceur CPU implémenté choisit la classe de service du domaine prêt ayant la priorité la plus élevée, puis ordonnance les processus au sein de cette classe. L'architecture étend ce modèle à des classes CPU, mémoire, E/S de stockage et E/S réseau indépendantes, permettant à un domaine de bénéficier d'un service sensible à la latence tandis qu'un autre consomme les capacités disponibles. L'application complète de la QoS entre ressources reste à réaliser. L'expérience établit la frontière architecturale et la première partie du modèle d'ordonnancement.

Une personnalité définit le contrat d'espace utilisateur visible dans un domaine : format exécutable, appels système, comportement des processus et signaux, sémantique VFS, périphériques, terminaux, horloges, réseau et attentes relatives aux espaces de noms. Linux x86-64 est la seule personnalité implémentée. BSD constitue une expérience plausible, car il pourrait réutiliser les mêmes mécanismes de domaine, mémoire, ordonnancement et périphériques, mais aucun ABI BSD n'a été construit ou testé. Les espaces de noms Linux pourraient à terme subdiviser un domaine Linux ; ils resteraient à l'intérieur de la frontière externe, plus forte, du domaine.

Les racines des domaines combinent des couches OCI vérifiées avec une surcouche inscriptible locale au démarrage. Un volume persistant détenu par l'hôte conserve la configuration du contrôleur et un cache OCI adressé par contenu entre les redémarrages. Les comportements sélectionnés de tmpfs, overlay, procfs, devfs et du volume persistant sont implémentés ; le vaste écosystème de systèmes de fichiers Linux et les garanties de stockage de production restent hors du périmètre du prototype.

4.4 Compatibilité Linux, identités et autorité

La seule couverture des numéros d'appels système Linux renseigne peu sur la conformité POSIX. POSIX est principalement un contrat source et comportemental couvrant libc, processus, fichiers, signaux, terminaux, horloges, synchronisation et IPC. Les espaces utilisateur musl et glibc évalués exercent fork/exec/wait, opérations sur chemins et descripteurs, tubes et FIFO, signaux, horloges, termios, sockets, threads fondés sur futex et certains IPC POSIX et System V. Linux n'offre qu'une couverture partielle et n'a pas encore exécuté une suite complète de conformité POSIX.

4.4 Compatibilité Linux, identités et autorité (suite)

Chaque domaine possède une table native de capacités scellée. Une capacité désigne un objet noyau et accorde une combinaison de droits de lecture, écriture, gestion et exécution. L'autorité initiale provient uniquement de l'enveloppe de démarrage ; la délégation peut réduire les droits, mais pas les étendre ; les handles sont étiquetés par génération et la révocation de la racine supprime les droits dérivés. Les périphériques, pages DMA, files, volumes et opérations de gestion peuvent donc être accordés indépendamment. Les classes de service CPU sont aujourd'hui appliquées par cette frontière de domaine ; l'application complète de la contention mémoire et E/S reste inachevée.

À l'intérieur de cette frontière, la personnalité Linux applique des identifiants et permissions propres au domaine. Le profil évalué mappe UID/GID 0 vers l'identifiant noyau 100000 et utilise les identifiants réel/effectif/sauvegardé dans les contrôles de signaux, interprocessus et d'accès aux fichiers. Le VFS implémente les opérations de mode, propriété, umask et accès utilisées par le chemin du gestionnaire de paquets. Le profil ne comporte qu'une identité mappée, aucune capacité Linux et refuse de nombreuses mutations d'identifiants. Les ACL POSIX et les cas limites de permissions restent incomplets ; aucun SELinux ni autre module de sécurité Linux actif n'applique d'étiquettes. L'UID Linux 0 ne peut pas contourner la table native des capacités.

La conception établit le mécanisme nécessaire à RQ1 ; la section suivante évalue la quantité de comportement Linux effectivement prise en charge.

5 Évaluation

L'évaluation combine quatre formes de preuve : des charges Linux progressivement moins synthétiques, un registre des appels système et interfaces, un démarrage systemd composé et des flux de travail opérateur, ainsi qu'un harnais de performance contrôlé mais non équivalent à un seul vCPU. Ces mesures répondent à des questions différentes. Un démarrage réussi démontre l'intégration, le registre consigne l'étendue et les lacunes, les flux interactifs exposent le comportement du cycle de vie et le benchmark identifie des défauts de performance ; aucun de ces éléments ne suffit à établir une compatibilité Linux générale.

5.1 Résultat fonctionnel et périmètre de recherche

Le prototype démarre une image EFI x86-64 unifiée, exécute son contrôleur en anneau 3, obtient un accès réseau sous QEMU et télécharge puis met en cache des systèmes de fichiers OCI. Il exécute des programmes Alpine musl/BusyBox et Rocky glibc, lance systemd Rocky jusqu'à multi-user.target et conserve des états indépendants de processus, espaces d'adressage, racines, montages et cycle de vie entre plusieurs domaines.

Le registre classe les 375 numéros d'appels système Linux 6.12 x86-64 : 82 complets, 245 partiels et 48 refusés par politique. Chaque numéro est classé, mais seuls 82 sont consignés comme complets ; « aucun manquant » dans le registre signifie couverture de distribution, et non conformité. Les chiffres montrent l'étendue, pas l'exhaustivité. Le chargement et le comportement des processus, les pseudo-systèmes de fichiers, cgroups, espaces de noms, IPv6, familles ioctl, systèmes de fichiers et périphériques restent incomplets. Il n'existe ni écosystème GPU, ni prise en charge matérielle étendue, ni ABI stable de pilotes tiers, ni audit de sécurité indépendant, ni référence de performance native, ni garantie d'exécution générale des applications Linux.

Ces lacunes définissent la frontière de recherche. Le résultat démontre que l'IA peut construire suffisamment de noyau pour exécuter les charges Linux testées, et non qu'une application ou distribution Linux arbitraire fonctionnera, ni que Linux peut remplacer des décennies d'ingénierie Linux.

5.2 Performances et benchmarking

Les tests de performance ont commencé comme une vérification diagnostique de la réalité, et non comme une affirmation d'équivalence. Le harnais à un seul vCPU fixait la machine QEMU, la mémoire, le modèle de CPU, l'accélérateur, les commandes de charge et les limites de mesure. Deux expériences distinctes ont été réalisées et sont rapportées séparément ci-dessous.

5.2 Performances et benchmarking (suite)

L'expérience de démarrage mesurait le temps écoulé entre le lancement de QEMU et le moment où systemd indiquait avoir atteint la cible multi-user. Rocky sur Linux utilisait l'image officielle Rocky Linux 10.2 GenericCloud, démarrée localement depuis qcow2 avec son noyau Rocky standard, version 6.12.0-211.16.1.el10_2.0.1 pour x86-64. Rocky sur Ginix utilisait l'image OCI Rocky Linux 10 UBI-init. Il s'agit de profils d'espace utilisateur, de stockage et de démarrage différents. Leur point d'arrivée commun rend le résultat intéressant sur le plan opérationnel, mais il ne s'agit pas d'une comparaison contrôlée des performances de démarrage des noyaux.

L'expérience de charge exécutait le même exécutable sysbench Rocky Linux 10 x86-64, vérifié par empreinte, dans les deux invités. Le côté Linux utilisait le système GenericCloud préparé. Le côté Ginix utilisait une racine Rocky 10 volontairement minimale contenant /bin/sh, /usr/bin/sysbench et leurs dépendances d'exécution ; elle n'utilisait pas la racine systemd UBI-init. La mesure de charge excluait le démarrage de l'invité et la préparation de l'image. Le fait d'apparier l'exécutable, les commandes et les paramètres QEMU réduit des variations importantes, mais les espaces utilisateur, services, chemins de stockage et noyaux environnants restent différents.

Les premiers résultats étaient particulièrement mauvais. Le CPU était déjà proche du résultat Rocky, mais la mémoire n'atteignait que 48,390 MiB/s contre 679,510 MiB/s pour Rocky, et le test de threads ne réalisait que 0,078 événement/s contre 306,129 — environ 14 fois et 3 900 fois plus lent, respectivement. Le benchmark est alors devenu un contrôle d'acceptation. Le responsable du projet a exigé des entrées appariées, clarifié le périmètre à un seul vCPU et demandé à Codex de profiler, modifier le noyau, relancer les mesures et ne conserver que les améliorations vérifiées. Dans ce cadre, Codex a choisi l'instrumentation et les changements d'implémentation, puis a itéré sur des exécutions QEMU répétées. Il s'agissait de la même procédure HDAE que celle décrite à la section 3, appliquée à un problème de performance plutôt qu'à un jalon fonctionnel.

Au cours des heures suivantes, les mesures ont révélé des défauts précis plutôt qu'un noyau généralement lent. Le chemin des threads sélectionnait à plusieurs reprises un thread superviseur venant d'être préempté, puis sauvegardait et restaurait un contexte utilisateur complet alors qu'aucun autre thread ne pouvait s'exécuter. Le test mémoire était dominé par le coût des horodatages : ses opérations de 1 Kio entraient constamment dans le noyau parce que Ginix ne disposait pas de vDSO. La sélection de l'ordonnanceur, les yields sans effet, la trace diagnostique, la validation des tables de pages et le chemin d'horloge protégé ont été resserrés un par un, les régressions étant rejetées par la suite de vérification existante. Un vDSO compatible Linux et une page vvar en lecture seule ont ensuite assuré une mesure rapide du temps pendant le processus initial et les execve() ultérieurs. Une seconde passe a rendu l'instantané vvar immuable pour chaque image d'exécution et remplacé la division d'horloge par une conversion en virgule fixe et une normalisation réciproque en temps constant.

La médiane du test de threads est passée de 0,078 à 309,732 événements/s, et le débit mémoire de 48,390 à 620,580 MiB/s. Une boucle de profilage antérieure avait également réduit le chemin interne Ginix entre Linux-init et la console, de 15,344 s à une plage de 4,08-4,45 s, en supprimant un blocage dû au polling, des balayages de trames quadratiques et la copie anticipée de pages immuables du système de fichiers racine. L'historique des optimisations est consigné dans le dossier de performance.

Chemin de démarrage	Échantillons	Médiane jusqu'à la cible	Rapport observé
GenericCloud sur Linux	3	42,579 s	3,31× le temps Ginix
UBI-init sur Ginix	3	12,860 s	référence

Tableau 2 : temps exploratoire jusqu'à multi-user.target sous QEMU TCG x86-64 sur un hôte Apple Silicon. Les chemins de démarrage ne sont pas équivalents.

Charge	Mesure	Rocky sur Linux	Rocky sur GnuX	Différence observée
CPU	événements/s	232,360	267,650	GnuX +15,2 %
Mémoire	MiB/s	679,510	620,580	GnuX –8,7 %
Threads	événements/s	306,129	309,732	GnuX +1,2 %

Tableau 3 : médiane de neuf observations par système pour des charges sysbench de trois secondes sous QEMU TCG x86-64. Une valeur plus élevée est préférable. L'exécutable et les commandes correspondent ; les environnements d'exécution complets, eux, ne sont pas identiques.

Les résultats des charges racontent deux histoires circonscrites. Premièrement, le CPU et le yield mono-thread ont atteint une quasi-parité dans cet environnement traduit. Lus avec prudence, ces chiffres montrent que les écarts les plus importants provenaient de défauts d'implémentation qu'une boucle HDAE guidée par benchmark pouvait détecter et réduire substantiellement. L'écart CPU de 15,2 % peut refléter la variation de TCG ou des tâches d'arrière-plan différentes ; il ne prouve pas que GnuX exécute généralement les charges CPU plus rapidement. Deuxièmement, l'ajout de la mesure du temps vDSO/vvar compatible Linux a fait passer le débit mémoire de 48,390 à 620,580 MiB/s, laissant Rocky sur GnuX 8,7 % sous Rocky sur Linux pour le test de 1 Kio. La majeure partie de l'écart initial provenait du coût des horodatages plutôt que de la copie brute ; cette expérience TCG ne permet pas d'isoler la cause de la différence résiduelle.

Le résultat de démarrage compare séparément deux chemins de démarrage. GnuX restaure un système de fichiers mis en cache et lance directement un domaine. Rocky démarre depuis un disque qcow2 local via son UEFI généraliste, son noyau, son initramfs et son chemin de découverte des périphériques. Le rapport mesuré décrit ces chemins complets, y compris le travail omis par l'un d'eux, plutôt que les performances globales des noyaux.

Ces mesures TCG montrent que GnuX peut exécuter le même benchmark Linux non trivial et que les itérations HDAE peuvent transformer des défauts catastrophiques en médianes assez proches pour cette suite étroite. Elles n'établissent pas de comparaison générale des performances des noyaux. Une expérience plus solide exigerait des profils d'espace utilisateur identiques, des états de stockage et de services équivalents, des exécutions plus longues, des dispersions et intervalles de confiance, ainsi que des mesures x86-64 KVM. Les performances réseau, IPC, multicœurs, stockage et sécurité n'ont pas été mesurées.

5.3 Artefact, vérification et reproductibilité

Le code source évalué est identifié par le commit d11c126a048a51fe7eb9c4105f31760add1027ea. Le dépôt contient l'architecture et les journaux de décisions, le registre de compatibilité, les harnais de test QEMU, le profil de benchmark, l'URL et l'empreinte de l'image Rocky épinglée, le générateur de racine sysbench GnuX et les cibles Make utilisées pour les contrôles rapportés. Les principales commandes se répartissent en trois groupes :

- vérification : make check et make test-rocky-systemd ;
- préparation : make benchmark-prepare-linux et make benchmark-prepare-gnuX ;
- mesure : make benchmark-boot-comparison, make benchmark-gnuX-sysbench et make benchmark-compare.

5.3 Artefact, vérification et reproductibilité (suite)

Au point de contrôle évalué, make check réussissait et le harnais systemd Rocky atteignait à plusieurs reprises sa cible de connexion vérifiée. La cible agrégée make test ne réussissait pas complètement : son démarrage de vérification échouait encore dans une sonde du cycle de vie du superviseur système après `execve()`. Cet échec connu fait partie du point de contrôle et empêche de décrire le dépôt comme disposant d'une suite de tests entièrement réussie.

La reproduction est également limitée par des preuves qui n'ont pas été conservées dans le commit évalué. Les journaux bruts de benchmark et les tableaux de résultats subsistants ont été archivés ultérieurement avec l'artefact de l'article, accompagnés de sommes de contrôle et d'une déclaration de périmètre. Le modèle exact de l'hôte, la version de macOS, la version de QEMU, l'utilisation des jetons exportée par le fournisseur et un export verbatim de la conversation n'ont pas été archivés. Les scripts versionnés et sorties brutes permettent l'inspection et une nouvelle exécution, mais ne permettent pas de reconstruire la trajectoire originale. Une publication devrait associer le commit évalué à ces résultats, aux versions exactes des outils et de l'hôte lorsqu'elles peuvent être retrouvées, à la sortie complète des commandes et à un manifeste de réussite/échec lisible par machine.

5.4 Exemples de flux de travail opérateur

Les transcriptions condensées suivantes illustrent l'expérience opérateur visée. Elles utilisent des commandes implémentées du contrôleur, mais omettent les longues sorties du registre, de systemd et des mesures temporelles. Ce sont des pseudo-captures d'écran illustratives, et non des journaux bruts de benchmark.

Exemple 1 : démarrer un espace utilisateur Linux conventionnel.

```
ginux> domain new rocky
rocky: created (image required)
ginux> domain set rocky image=rockylinux/rockylinux:10-ubi-init
ginux> domain set rocky init=/sbin/init memory-mib=2048
rocky: settings accepted; configured
ginux> domain start rocky --attach
ginux: OCI cache hit (2 layers)
ginux: starting Linux personality domain
systemd: Reached target Multi-User System.
Rocky Linux 10
rocky login:
```

Ce cas à domaine unique sélectionne une image d'espace utilisateur Linux et un processus init, tandis que Ginux demeure le noyau. Le contrôleur peut aussi gérer plusieurs domaines sans traiter une charge de travail particulière comme le système d'exploitation hôte.

Exemple 2 : réutiliser le contenu entre plusieurs domaines.

```
ginux> domain new alpine-a
ginux> domain set alpine-a image=alpine:latest init=/bin/sh
alpine-a: settings accepted; configured
ginux> domain start alpine-a
ginux: OCI cache miss; pulling 1 layer
alpine-a: running (detached)
ginux> domain new alpine-b
ginux> domain set alpine-b image=alpine:latest init=/bin/sh
ginux> domain start alpine-b
ginux: OCI cache hit (1 layer)
alpine-b: running (detached)
ginux> domain list
NAME    STATE  PERSONALITY INIT    NETWORK
rocky   running linux    /sbin/init host(shared)
alpine-a running linux    /bin/sh   host(shared)
alpine-b running linux    /bin/sh   host(shared)
ginux> shutdown
ginux: local configuration committed
```

Après redémarrage, le contrôleur récupère les définitions de domaines et les couches OCI mises en cache depuis le volume détenu par l'hôte. Au point de contrôle évalué, les surcouches inscriptibles des domaines restent locales au démarrage : la configuration et le cache d'images persistent, sans que cela implique la durabilité de chaque modification de charge de travail.

5.4 Exemples de flux de travail opérateur (suite)

Pris ensemble, le démarrage composé, le registre de compatibilité, les flux de travail et le benchmark limité répondent à RQ1 et RQ3 uniquement dans les limites évaluées. Linux exécute les charges de distribution évaluées et démontre que plusieurs défauts graves de performance pouvaient être corrigés au moyen de la même boucle HDAE. Le registre partiel et les mesures limitées empêchent de transformer ces résultats en affirmation générale de compatibilité ou de performance Linux.

6 Menaces à la validité

Il s'agit d'une étude de cas observationnelle ayant réussi, et non d'une estimation contrôlée de la fiabilité des agents de programmation ni d'une validation de la supériorité générale de la HDAE. Le projet a utilisé une seule trajectoire de modèle et un dépôt évolutif. Il n'y a eu ni projet humain seul apparié, ni exécution avec un autre modèle, ni reconstruction indépendante à partir des exigences initiales. L'auteur a fourni l'architecture et jugé si le comportement observé était acceptable. Cela apportait un jugement éclairé, mais introduisait également un biais d'évaluation.

La taille du dépôt, les commits, les jours écoulés et le coût de l'API décrivent le projet, mais ne mesurent ni une quantité équivalente de travail humain ni la qualité du code. Le client a pu être mis à jour pendant l'étude, les totaux de jetons exportés par le fournisseur n'ont pas été archivés et le coût d'environ 8 000 USD est rapporté par l'auteur. La conversation, les plans, l'historique Git et les tests préservent une provenance substantielle, mais ne rendent pas la trajectoire exactement reproductible.

Les contrôles fonctionnels sont composés, mais incomplets. Démarrer systemd Rocky et exécuter des applications sélectionnées établit davantage que de simples stubs d'appels système, mais pas une conformité générale à Linux ou POSIX. Le contrôle complet make test conservait également l'échec du superviseur après `execve()` décrit plus haut. Les résultats de performance proviennent d'une configuration étroite à un seul vCPU, sous QEMU TCG, avec des profils de démarrage et d'espace utilisateur différents, des charges courtes, peu d'échantillons et sans résultats bruts archivés ; ils ne permettent aucune affirmation générale sur les performances.

Aucun audit de sécurité indépendant n'a été effectué. Les propriétés de sécurité au-delà des mécanismes testés restent en dehors des preuves rapportées ici.

7 Discussion

7.1 Alternative : commencer avec Linux

Un effort produit pratique aurait eu de bonnes raisons de commencer avec Linux. Les espaces de noms, cgroups, KVM, pilotes de périphériques, systèmes de fichiers, modules de sécurité et une vaste suite de compatibilité existent déjà. Un contrôleur de type Linux pourrait gérer un hôte Linux immuable, tandis qu'une expérimentation de domaines pourrait initialement combiner ces mécanismes et ajouter des changements d'ordonnancement, de périphériques et de politiques de ressources là où les abstractions existantes se révéleraient insuffisantes. Cette voie aurait permis d'obtenir beaucoup plus rapidement une prise en charge étendue du matériel et des applications.

Elle aurait aussi répondu à une autre question. Cette expérience a délibérément retiré Linux du côté noyau afin de tester si la HDAE pouvait construire elle-même les mécanismes et découvrir un objet externe d'isolation distinct. Commencer avec Linux déplacerait le travail de la réimplémentation vers l'intégration, le renforcement, la mesure et une conception de qualité amont. C'est peut-être l'implication la plus utile : si l'IA peut parcourir le chemin d'une construction à partir de zéro, une équipe peut plutôt la diriger vers des améliorations circonscrites d'un système mature, où l'écosystème reste intact et où l'idée nouvelle doit résister aux normes existantes de revue, de compatibilité et de performance.

7.2 Les connaissances du domaine et le jugement humain façonnent les résultats

Codex a raccourci le chemin entre une décision technique et du code fonctionnel ; les connaissances du domaine ont fourni la vision, les priorités et la norme de réussite. Ces connaissances étaient nécessaires pour choisir le contrat de l'espace utilisateur Linux, séparer le transport OCI de l'isolation à l'exécution, reconnaître un comportement fork/exec défectueux, définir la propriété des domaines, interpréter les échecs de systemd et distinguer une sortie convaincante d'un comportement correct.

L'orientation du projet s'appuyait sur l'expérience de Caos Linux, CentOS et Rocky Linux ; de l'écosystème de conteneurs Singularity/Apptainer ; des écosystèmes de provisionnement de clusters Warewulf et Perceus ; ainsi que des opérations d'infrastructure en production [11, 2]. Cette expérience a façonné le flux de travail du contrôleur, le cache persistant d'images, la sémantique de démarrage des domaines, les exigences de provisionnement, les contrôles d'acceptation et la décision d'arrêter. Les résultats apparemment réussis mais sémantiquement incorrects sont devenus des tests de régression ou des corrections architecturales.

L'IA offre également aux ingénieurs moins expérimentés un chemin plus court vers des contributions circonscrites et utiles : explorer du code inconnu, générer des tests, comparer des interfaces et implémenter des changements soumis à revue. L'expérience reste décisive pour choisir les abstractions, reconnaître les résultats incomplets et comprendre quand une réussite locale crée un risque pour l'ensemble du système. Les contrôles décrits à la section 3 rendent ce jugement durable. L'IA élargit le nombre de personnes capables de construire ; la responsabilité humaine détermine ce qui mérite la confiance.

7.3 Le code n'est pas le principal avantage concurrentiel

Construire un petit noyau fonctionnel est une tâche délimitée. Maintenir un noyau de production implique de prendre en charge des matériels et charges de travail variés, de préserver les interfaces, de réparer les défauts, de coordonner contributeurs et fournisseurs et de maintenir une cadence de publication fiable. Le code est relativement facile ; les flux de travail, la compatibilité, la maintenance et la communauté sont difficiles.

Linux a utilisé, selon l'estimation de l'auteur, environ 8 000 USD d'appels API. Ce chiffre décrit l'expérience, mais ne mesure ni la productivité, ni une quantité équivalente de travail humain, ni la qualité du code. Le contraste pertinent est plus restreint : un coût direct modeste en API a produit un prototype substantiel, tandis que la compatibilité, la confiance, l'intégration industrielle et la maintenance représentent des investissements bien plus importants.

Cette pression sera probablement plus forte au-dessus du middleware et de l'orchestration, où les logiciels peuvent être remplacés sans déplacer tout un écosystème matériel et systèmes. Les logiciels fondamentaux peuvent eux aussi être réimplémentés, mais l'adoption d'infrastructures dépend des normes, des interfaces, de la certification, de la prise en charge matérielle, de la continuité communautaire et du soutien commercial. La nouveauté a peu de valeur sans cet écosystème.

Linux en est l'exemple le plus clair. Sa valeur dépasse largement le code nécessaire pour démarrer un shell. Il fournit des attentes stables entre processeurs, périphériques, systèmes de fichiers, hyperviseurs, distributions, fournisseurs et applications. Linux ne possède ni matrice matérielle comparable, ni organisme de normalisation, ni communauté engagée à maintenir le fonctionnement des programmes d'aujourd'hui. L'IA peut construire des éléments techniques ; l'adoption exige encore que des organisations indépendantes soutiennent les mêmes contrats.

8 Conclusion

Pour RQ1, Linux fournit un résultat circonscrit : un processus HDAE centré sur un agent de programmation persistant unique a produit un noyau indépendant qui démarre certains programmes Alpine non modifiés et l'espace utilisateur systemd Rocky Linux évalué. Il prend également en charge des domaines natifs du noyau et a exécuté le même exécutable sysbench que celui utilisé sur un système Rocky standard de comparaison. Il s'agit d'une intégration rapide de mécanismes établis, et non de l'invention de l'ingénierie des noyaux.

Pour RQ2, la méthode décisive n'était ni un prompt unique ni une autonomie sans contrainte. Une architecture écrite, des contrôles comportementaux par étapes, une tâche persistante, des charges réelles et une interprétation humaine éclairée ont maintenu la boucle d'implémentation HDAE alignée sur l'objectif du système. Ce cas unique motive la méthode ; il ne valide pas la HDAE par rapport à d'autres processus. Pour RQ3, le résultat reste limité : la compatibilité Linux est partielle, les preuves de performance sont exploratoires, la cible de test complète conserve un échec connu et les exigences de production relatives au matériel, à la sécurité, aux normes et à la maintenance ne sont pas satisfaites.

L'IA peut produire du code profondément enfoui dans la pile systèmes. La valeur des infrastructures fondamentales découle de la compatibilité, de la prise en charge matérielle, de la revue, de la confiance communautaire et de la responsabilité à long terme. Linux a acquis ces qualités au fil de décennies d'ingénierie humaine couvrant une immense variété de charges de travail. Linux a reproduit une partie limitée du code, pas cette institution. Une implémentation moins coûteuse ne suffit pas, à elle seule, à justifier le remplacement d'une fondation à laquelle un écosystème fait déjà confiance.

Références

- [1] Nicholas Carlini. Building a C compiler with a team of parallel Claudes. Anthropic Engineering, 2026.
- [2] CIQ. The founding story of CIQ, from Berkeley Lab to Rocky Linux, 2026.
- [3] Jonathan Corbet. The 2026 maintainers summit. LWN.net, octobre 2026.
- [4] Jingzhe Ding, Shengda Long, Changxin Pu, et al. NL2Repo-Bench: Towards long-horizon repository generation evaluation of coding agents. In Proceedings of the 43rd International Conference on Machine Learning, volume 306 of Proceedings of Machine Learning Research, pages 25035–25054, 2026.
- [5] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world github issues? In International Conference on Learning Representations, 2024.
- [6] Thomas Kwa, Ben West, Joel Becker, et al. Measuring AI ability to complete long software tasks. METR research report, 2025.
- [7] Linux Kernel Project. Kernel guidelines for tool-generated content, 2026.
- [8] Alex Mathai, Chenxi Huang, Suwei Ma, et al. CrashFixer: A crash resolution agent for the Linux kernel. arXiv preprint arXiv:2504.20412, 2025.
- [9] Nift Contributors. Ai development, 2026.
- [10] Adam Payne. AgentBubble: Human-directed agentic engineering. GitHub repository, 2026.
- [11] Rocky Linux. About Rocky Linux, 2026.
- [12] Minh V. T. Thai, Tue Le, Nghi D. Q. Bui, et al. SWE-EVO: Benchmarking coding agents in long-horizon software evolution scenarios. arXiv preprint arXiv:2512.18470, 2025.
- [13] UEFI Forum. UEFI Specification 2.11: Overview. UEFI Forum specification, 2024.