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

AutoCluster: Un agent pour le clustering basé sur les grands modèles de langue

Erwan Versmée[1, 2] Youcef Remil[2] Mehdi Kaytoue[2] Julien Velcin[1]
(1) Université Lumière Lyon 2, Université Claude Bernard Lyon 1, ERIC, 69007, Lyon, France
(2) Infologic, 99 Av. de Lyon, 26500 Bourg-lès-Valence, France
e.versmee@univ-lyon2.fr, yre@infologic.fr, mka@infologic.fr,
julien.velcin@univ-lyon2.fr

RÉSUMÉ

Cette recherche présente AutoCluster, un agent basé sur les grands modèles de langue pour des tâches de classification non supervisée. Nous concevons trois agents dont deux sont basés sur la littérature et l'un, AutoCluster, est une contribution originale. Une analyse détaillée de leur performance sur 26 jeux de données de clustering révèle la supériorité de notre agent par rapport aux solutions de l'état de l'art. Enfin, nous justifions l'efficacité de notre agent à travers les nombreuses améliorations empiriques apportées au fur et à mesure de son développement.

ABSTRACT

AutoCluster : an LLM-based clustering agent

This work unveils AutoCluster, a Large Language Models based agent for unsupervised classification tasks. Three agents are constructed, from which two are heavily based on state-of-the-art and one, AutoCluster, is an original contribution. Evaluations on 26 clustering datasets demonstrate the superiority of our approach, which we explain through empirical improvements that have been brought throughout the development process.

MOTS-CLÉS : Agent basé sur les LLMs, clustering, science des données, ReAct, appel d'outils.

KEYWORDS: LLMs-based agent, clustering, data science, ReAct, function calling.

1 Introduction

L'IA générative est aujourd'hui basée en grande partie sur les grands modèles de langue (*Large Language Models*, LLMs). Excellant déjà dans des tâches classiques de TAL comme la traduction ou la réponse aux questions (Brown *et al.*, 2020; Bommasani *et al.*, 2021), ceux-ci jouent désormais un rôle clé dans des applications avancées telles que l'assistance à la rédaction de code ou l'analyse de documents. L'émergence de ces modèles a ouvert la voie à une nouvelle génération d'*agents* capables de raisonner, planifier et interagir de façon autonome avec "l'extérieur" (Yao *et al.*, 2023b; Shinn *et al.*, 2023). La littérature (Xi *et al.*, 2025) décrit un agent à travers trois modules : la *perception*, qui lui permet de connaître son environnement, l'*action* afin d'interagir avec ce même environnement, et le *cerveau*, équipé de mémoire et de connaissances, contrôlant raisonnement et décisions. Les agents basés sur les LLMs sont de tels agents, dans lesquels un LLM a le rôle de module cérébral. Ces derniers ont, entre autres choses, été popularisés grâce aux systèmes de RAG (Retrieval Augmented Generation) (Lewis *et al.*, 2020; Guu *et al.*, 2020), démontrant leur applicabilité et ouvrant la voie

à un large éventail d'applications, notamment dans l'industrie. Cet article explore spécifiquement l'utilisation d'agents — désignant dans la suite de cet article agents basés sur des LLMs — pour des tâches d'analyse de données, en particulier des tâches de regroupement non supervisé (clustering).

Ces dernières années, de nombreux agents dédiés à la science des données ont émergé. Certaines approches tentent d'automatiser des processus entiers d'analyse de données (Hong *et al.*, 2024; Li *et al.*, 2024; Guo *et al.*, 2024; Grosnit *et al.*, 2024; Shen *et al.*, 2023; Huang *et al.*, 2024b). Elles sont très largement basées sur les capacités de planification des LLMs. Pourtant, ces capacités sont remises en cause par la littérature récente (Kambhampati *et al.*, 2024; Bouzenia *et al.*, 2024; Xia *et al.*, 2024; Huang *et al.*, 2024a). Nous suivons des idées de ces derniers articles pour notre contribution, en diminuant l'autonomie et la prise de décision de l'agent que nous construisons (cf. 3.2.3).

La création de collections de référence pour évaluer ces agents est également un domaine de recherche émergent et en pleine expansion. PyBench (Zhang *et al.*, 2024) inclut mais ne se limite pas à des tâches d'analyse de données, tandis que MLE-Bench (Chan *et al.*, 2025) et DSBench (Jing *et al.*, 2025) se concentrent davantage sur le machine learning, s'appuyant respectivement sur 75 et 74 compétitions Kaggle (une plateforme de référence pour le partage et l'analyse de données). Ces deux derniers benchmarks, bien que pertinents, présentent une limite : ils comparent les résultats des agents non pas au jeu de test privé des compétitions, mais à un jeu de validation issu du jeu d'entraînement, rendant les évaluations moins fiables. MLAgentBench (Huang *et al.*, 2024b) définit un benchmark plus restreint également basé sur Kaggle et évalue la performance d'un agent en le comparant à une baseline relativement faible. InfiAgent-DABench (Hu *et al.*, 2024) évalue des agents sur des questions d'analyse de données spécifiques et délimitées, par opposition à une évaluation bout-à-bout d'un flot de travail complet. Néanmoins, aucun de ces benchmarks ne se concentrent sur le clustering, ou sur des tâches d'analyse de données non supervisées.

La fouille de données constitue un aspect essentiel de l'analyse de données, avec de nombreuses applications dans le monde industriel, notamment en software engineering (Remil *et al.*, 2021). Parmi les différentes approches existantes, le clustering est une technique permettant de rassembler des données similaires dans des mêmes groupes de façon non supervisée. Cela permet notamment d'obtenir une idée de la structure sous-jacente des données sans avoir besoin de les annoter. Les recherches de la littérature récente sur les agents s'intéressent principalement aux tâches supervisées et beaucoup moins des problèmes non supervisés.

Dans cet article, nous construisons, comparons et évaluons différentes approches à base d'agents pour l'exécution de tâches de clustering. Les approches comparées proviennent de la littérature hormis une, **AutoCluster** [1], que nous avons construite spécifiquement pour cette tâche. Elles diffèrent par leurs architectures, c'est-à-dire la façon dont sont formulés et articulés les prompts (instruction textuelle guidant ou contraignant le modèle), et dans le degré d'autonomie laissé aux LLMs. Les architectures utilisées sont, en théorie, agnostiques aux modèles de langue, ce qui facilite la comparaison des performances de différents LLMs. Les modèles utilisés lors des expériences sont en open source ou mis à disposition gratuitement via des API (dans la limite des taux d'accès définis par les fournisseurs).

La contribution de cet article est triple :

- Nous développons un agent spécialisé dans le clustering, inspiré de la littérature récente et évaluons sa performance sur 26 jeux de données adaptés.
- Nous explorons la performance d'agents de la littérature pour des tâches d'analyse de données en mode non supervisé, en particulier pour du clustering.

1. https://anonymous.4open.science/r/AutoCluster-DE39

— Nous détaillons les difficultés que rencontrent des agents pour ce type de tâches, et discutons des diverses approches pour y pallier.

2 Travaux connexes

De nombreuses recherches récentes se sont focalisées sur les différentes stratégies de prompting d'un LLM afin d'optimiser la génération (Brown *et al.*, 2020; Wei *et al.*, 2022; Besta *et al.*, 2024). Ces modèles rencontrent souvent des difficultés face à des tâches complexes nécessitant plusieurs étapes, un raisonnement itératif ou des calculs avancés (Ma *et al.*, 2024). Sur la base de cette analyse, de nouvelles directions de recherche ont émergé dans le domaine des agents. Plusieurs architectures ont vu le jour, propulsées par l'engouement lié aux LLMs, dans le but d'optimiser l'entrelacement entre le raisonnement du LLM et l'utilisation d'outils externes (Yao *et al.*, 2023b; Shinn *et al.*, 2023; Schick *et al.*, 2023).

Dans le cadre de cette étude, nous nous intéressons au clustering comme cas d'usage pour explorer et comparer différentes architectures de prompt issues de la littérature. En particulier, des architectures spécifiques ont déjà été développées dans des travaux connexes pour des tâches d'analyse de données : (Li *et al.*, 2024; Hong *et al.*, 2024; Guo *et al.*, 2024; Huang *et al.*, 2024b) et certaines méthodes ont été spécifiquement étudiées pour incorporer des LLMs dans des chaînes de traitement effectuant du clustering de textes : (Viswanathan *et al.*, 2024; Zhang *et al.*, 2023).

DataInterpreter (Hong *et al.*, 2024) repose sur une stratégie de planification hiérarchique et dynamique basée sur les LLMs, qui lui permet de décomposer un problème d'analyse de données en sous-tâches simples. Il utilise des outils pré-construits par des humains (snippets de code) ou par lui-même, intégrant un module "mémoire" qui enregistre ses propres solutions pour réutilisation ultérieure. Couplé à un module de vérification automatisée, dans lequel un LLM génère ses propres tests unitaires, cet agent est fortement dépendant de la capacité de planification et prise de décision des LLMs. Néanmoins, (Huang *et al.*, 2024b) met en évidence les limites à long terme de ces capacités. Les expériences qu'ils ont menées avec leur agent montre des taux de succès de 0% sur des tâches nouvelles (apparues après le pré-entraînement des LLMs utilisés). *AIDE* (Jiang *et al.*, 2025), conçu pour du ML, repose également sur de la planification par LLM, construisant dynamiquement un arbre de solutions potentiels, dans lequel un nœud contient une solution, et une branche correspond à un axe d'amélioration du parent. Bien qu'affichant des performances comparables aux humains, cette approche exige d'importantes ressources pour explorer plusieurs pistes en parallèle et peut rapidement devenir complexe à mesure que l'arbre s'étend.

DS-Agent (Guo *et al.*, 2024) tire profit d'une riche base de données, construites sur des compétitions Kaggle, fonctionnant sur un raisonnement à partir de cas. Des cas similaires (requêté par similarité sémantique) à un problème donné viennent enrichir le contexte fourni au LLM, améliorant la qualité du plan généré. Néanmoins, ce contexte supplémentaire n'est pas facilement accessible dans des scénarios du monde réel, et limite l'application de l'agent aux tâches présentent dans la base de données. AutoKaggle emploie de son côté un cadre multi-agent dans un flot construit pour l'analyse de données. Spécialisé pour Kaggle, le système suppose une structure classique de compétition, notamment avec un découpage des données en jeu de test et jeu d'entraînement, ce qui n'est pas adapté à une tâche non supervisée comme le clustering. De plus, cette approche utilise une surcouche d'outil qui englobent les outils classiques de Python pour l'analyse de données (notamment scikit-learn), ajoutant une strate de complexité supplémentaire pour l'agent, l'empêchant d'utiliser sa

connaissance paramétrique des librairies de machine learning.

Enfin, Agent K v1.0 (Grosnit *et al.*, 2024) emploie différents mécanismes des architectures cités ci-dessus, et orchestre des flots de travail pour des tâches de science des données. Comme (Guo *et al.*, 2024) et (Wang *et al.*, 2023a), cet agent exploite les capacités inhérentes des LLMs à raisonner à partir de cas. Il sélectionne activement les cas les plus pertinents en fonction de deux critères : la similarité avec la tâche en cours et la difficulté de résolution de l'exemple choisi. D'abord placé dans une machines à états finis (Xia *et al.*, 2024) pour découvrir et pré-traiter les données, le paradigme change pour la partie de mise en œuvre des modèles, dans laquelle l'agent effectue ses propres choix. À travers des outils d'analyse de données tels que AutoML (données tabulaires de ML) et HEBO (tuning des hyperparamètres), ou des prompts spécifiques, cet agent peut adopter des approches variées pour couvrir un maximum de diversité dans les problèmes rencontrés. Guidé par des méta test génériques, l'agent peut également décider, à n'importe quelle étape du processus, de prendre une action *intrinsèque*, c'est à dire sans action sur l'environnement (formuler une hypothèse, raisonner, etc.)

Ainsi, son rôle est plus celui d'un orchestrateur qui décide de la marche à suivre en fonction du problème. Bien que démontrant de fortes capacités (niveau de grand maître sur Kaggle) dans diverses tâches (tabulaires, vision, traitement du langage naturel, multimodales), cet agent présente certaines limites. Entre autres, il requiert un haut niveau de customisation des prompts par type de tâche (pour la partie de choix du modèle), difficile à mettre en place et à affiner. De plus, dû à son large éventail d'applications, il repose sur la capacité de planification des LLMs, qui est notamment mise en cause par (Kambhampati *et al.*, 2024; Liu *et al.*, 2023; Xia *et al.*, 2024).

En effet, d'autres volets de la littérature examinent dans quelle mesure les LLMs possèdent les compétences nécessaires pour être utilisés dans des applications de telle complexité (Huang *et al.*, 2024a; Kambhampati *et al.*, 2024). Des travaux récents montrent l'incapacité de ces modèles à planifier à long terme (Liu *et al.*, 2023), et à s'auto-critiquer de manière intrinsèque. Sans l'aide de sources externes, les LLMs ne parviennent pas à détecter *quand* une autocritique est nécessaire (c'est à dire lorsque leur sortie contient une erreur). Ainsi, solliciter l'auto-évaluation de ces modèles sans directives adéquates (comme pourrait en donner un oracle humain) entraîne une baisse drastique des performances. Poussant l'analyse plus loin, Agentless (Xia *et al.*, 2024) et RepairAgent (Bouzenia *et al.*, 2024) s'éloignent des systèmes agentiques populaires. Leur "agent" opère dans le cadre d'une machine à états finis plutôt que de faire ses propres choix (ou transitions) comme c'est généralement le cas dans d'autres approches. Nous nous inspirons de ces travaux pour le développement de AutoCluster.

Les agents présentés ci-dessus sont destinés à des usages relativement généraux. Ils sont conçus pour s'attaquer à la fois à des tâches fermées (qui possèdent une solution clairement prédéfinie) et ouvertes (qui n'ont pas une seule réponse correcte, encourageant la réflexion et la créativité). Le clustering se distingue comme une tâche à la fois ouverte et complexe qui, malgré sa pertinence, reste peu explorée dans l'évaluation de ces agents. C'est pourquoi nous choisissons de nous concentrer sur l'étude de leur capacité sur des tâches de clustering.

3 Approche proposée

Afin d'évaluer dans quelle mesure un LLM peut être sollicité pour des tâches relativement complexes d'analyse de données, nous avons construit, testé et comparé plusieurs architectures d'agents pour

traiter des tâches de clustering, en s'appuyant sur la littérature récente. Après une présentation des données (Section 3.1), nous expliquons dans 3.2 l'organisation interne des agents comparés (fonctionnement du "cerveau") avant de détailler le fonctionnement de leur module d'action (section 3.3), identique à chacun, qui leur permet d'interagir avec leur environnement (Xi *et al.*, 2025). Le module de perception des agents n'est pas présenté, car les données utilisées sont exclusivement textuelles ; le caractère multimodal de certains des LLMs testés n'est pas employé dans le périmètre de cet article.

3.1 Données

Pour nos expériences, nous utilisons 26 jeux de données synthétiques disponibles gratuitement sur github [2]. Ces jeux représentent des points dans l'espace en deux ou trois dimensions, déjà regroupés en clusters de formes plus ou moins complexes (anneaux, spirales, losanges, ...) avec plus ou moins de chevauchement inter-clusters. Ils sont filtrés pour ne garder que ceux au format csv ; les deux (ou trois) premières colonnes correspondent aux coordonnées dans l'espace, et la dernière à l'indice du cluster attribué pour ce point. Le nombre de clusters varie de deux à vingt pour celui en contenant le plus ; en moyenne, les jeux comportent environ 1250 coordonnées. L'intérêt de ces jeux est double :

— *Visualisation intuitive* : d'abord, la séparation visuelle des clusters est assez nette pour un humain (quand bien même les clusters se chevauchent), il est donc facile d'estimer si un algorithme de clustering a réussi à séparer les données.

— *Simplicité* : ensuite, les données sont relativement simples (coordonnées en deux voire trois dimensions) et ne nécessite pas d'extraction de caractéristique, comme ce que l'on peut trouver dans des problème d'analyse de données, ce qui permet d'isoler l'objectif de nos agents sur le clustering.

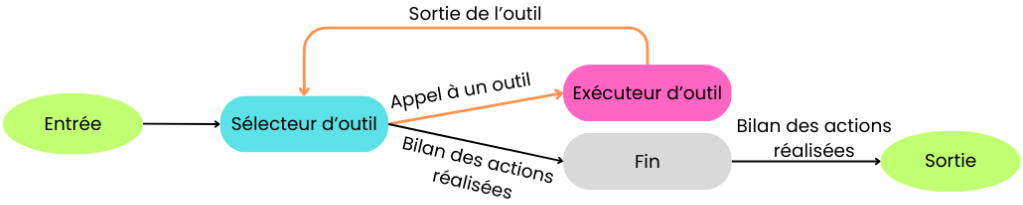
3.2 Architectures

Chacune des architectures présentées ci-dessous utilise des acteurs (sélecteur d'outil, analyste, planificateur, etc.) prenant part au processus. Ces acteurs sont implémentés par un appel à un modèle de langue avec un prompt personnalisé (cf figure 3) (en plus de l'historique de conversation) expliquant le rôle de l'acteur et indiquant les actions réalisables. Une différence importante entre les architectures provient donc de ces prompts permettant de spécialiser un LLM dans une tâche via de l'apprentissage en contexte (Brown *et al.*, 2020). Toutefois, ces dernières se distinguent également par le degré d'autonomie accordé aux modèles dans la prise de décision. Afin de garantir une comparaison équitable avec notre agent AutoCluster, et pour les adapter à notre tâche, les architectures issues de la littérature ont été réimplémentées avec LangChain (Chase, 2022). Enfin, chaque agent possède à sa disposition les mêmes outils, et les mêmes données en entrée.

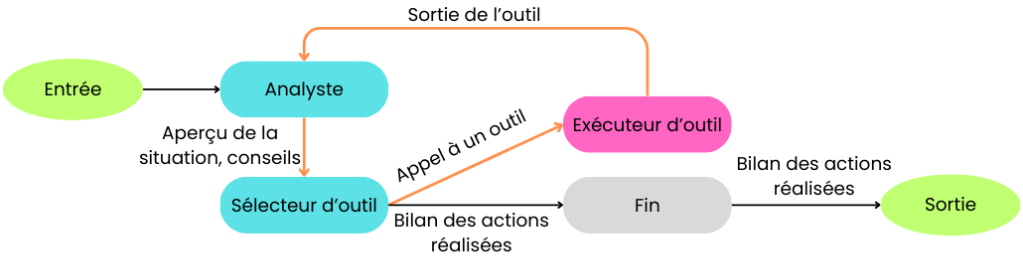
3.2.1 ReAct Clusterer

La première architecture utilisée est directement inspirée du cadre ReAct (Yao *et al.*, 2023b) (Reasoning and Acting) et l'agent associé est conformément nommé ReActClusterer. Cette architecture place un LLM dans une boucle, alternant successivement une étape de *raisonnement* et une étape d'*action*, tirant profit de deux paradigmes d'utilisation des modèles de langue. Elle bénéficie, d'une part, de leur capacité de raisonnement, mise en évidence par des recherches telles que (Wei *et al.*,

2. Jeu de données disponibles à : https://github.com/milaan9/Clustering-Datasets?tab=readme-ov-file



(a) ToolChain



(b) ThinkThenAct

FIGURE 1 – Comparaison des architectures basées sur ReAct. En bleu les appels à un LLM, en rose l'exécution d'un outil.

2022; Yao *et al.*, 2023a), et d'autre part de leur aptitude à agir, comme en témoigne leur capacité à appeler des outils (Schick *et al.*, 2023), ou réaliser des tâches de planification et décomposition en sous-tâches (Wang *et al.*, 2023a). Cette architecture relativement simpliste va permettre d'évaluer dans quelles mesures des approches plus complexes sont réellement nécessaires pour accomplir la tâche définie. Nous implémentons deux variantes de cet agent :

— **ToolChain** (TC), Figure 1 (a) : à chaque itération, l'agent doit choisir entre appeler un outil (bulle "Sélecteur d'outil"), ou bien terminer le processus en fournissant une explication des étapes réalisées (auquel cas il n'appelle pas d'outil).

— **ThinkThenAct** (TTA), Figure 1 (b) : identique à (a), à l'exception qu'une étape supplémentaire de pensée (bulle "Analyste" dans la figure) est insérée avant chaque appel à un outil. Plus coûteuse en tokens, cette variante offre néanmoins la possibilité à l'agent de se recentrer sur son objectif, augmentant potentiellement son efficacité.

Pour chacune de ces architectures, la décision de terminer le processus (passage dans le nœud Fin) est implicite dans le sens où le modèle n'a pas une action particulière à effectuer mais doit simplement ne plus appeler d'outil. Cela simplifie notamment l'implémentation, mais réduit également le nombre de tokens nécessaire pour prendre cette décision (l'agent n'a pas besoin de respecter un format particulier, ou d'appeler un outil pour terminer le processus).

Les nœuds en bleu représentent des appels au LLM, et les flèches des transitions d'un état à un autre. À chaque transition, un nouveau message est généré (soit du LLM, bulles en bleu, soit d'un outil, bulle en rose figure 1). Ces messages sont intégrés dans une variable, l'historique H, suivant un processus spécifique décrit ci-dessous.

Le processus reçoit en entrée un prompt de l'utilisateur définissant l'objectif de l'agent. En complément, il reçoit également des méta-données (instructions supplémentaires sur les librairies à utiliser, chemin d'accès aux données, etc.). Cette entrée initiale forme le premier message de l'agent, appelé T, qui lui sera fourni au début de chaque appel. Plus formellement, à chaque étape n d'appel à un

LLM (bulles bleues), l'agent émet une sortie S_i :

$$S_i \sim \pi(T, H, p) \tag{1}$$

avec p un prompt spécifique à chaque nœud indiquant l'objectif de l'agent à l'étape courante (voir les labels de transition partant des nœuds bleus figure 1), H un historique variant, composé d'une liste de messages ordonnés (décrit plus en détail ci-dessous), et π la politique générée par le LLM. Le contexte d'entrée du LLM (T, H, p) est ordonné du message le plus ancien (T) au plus récent (p).

Ces architectures sont légèrement adaptées afin d'obliger l'agent à re-rentrer dans la boucle si la sortie de l'exécuteur S_{exec} (bulles rose figure 1) est en erreur. Cela engendre une nouvelle itération (boucles orange figure 1) dont les entrées comprennent la tâche originale T, un résumé de la dernière étape réalisée avec succès $e_{succès}$ le cas échéant, et un prompt décrivant la nouvelle sous-tâche t_i de l'agent, qui consiste à corriger S_{exec} étant donné la cause de l'erreur c extraite de S_{exec} et les entrées E_{exec} de l'exécuteur (émises par la bulle bleue "Sélecteur d'outil" figure 1) qui ont conduit à l'erreur. On intègre également C et E_{exec} dans H. Sur l'équation 1, on a donc $p = t_i$ et $H = (e_{succès}, E_{exec}, c)$, avec H une liste *ordonnée* du plus ancien au plus récent.

Ce processus continue tant que la sortie de l'exécuteur est en erreur, afin de permettre les corrections nécessaires. Pour des erreurs en cascade, les sous-tâche précédentes (en erreur) $t_0 \dots t_{i-1}$ sont réécrites afin d'inclure à chaque étape la modification appliquée par l'agent et la cause de son erreur, cela permettant d'éviter à l'agent de répéter des erreurs faites précédemment. La nouvelle sortie $S_{répare}$ d'une bulle bleue est donc :

$$S_{répare} \sim \pi(T, e_{succès}, t_0, \dots, t_i) \tag{2}$$

avec $t_i \sim (E_{exec}, c)$ et $H = (e_{succ}, t_0 \dots t_{i-1})$, toujours ordonné du plus ancien au plus récent. Pour ThinkThenAct (figure 1 b), l'historique H du "Sélecteur d'outil" inclut en plus le message de l'Analyste.

Une fois sortie des erreurs, le contexte est mis à jour pour retirer les éléments liés à des erreurs et n'inclure que la dernière sortie de l'exécuteur en succès. On obtient un nouvel historique $H = (T, e_{succès})$. Ci-avant, $e_{succès}$ est un message formulé à partir d'une exécution en succès d'un outil, indiquant les entrées et la sortie de l'exécuteur (bulle rose). Le processus se modélise donc par une chaîne de Markov dont les états sont décrits ci-dessus et les transitions sont prédéfinies (on revient toujours à un LLM après l'exécution d'un outil) ou assurées par le LLM, qui agit ainsi comme un générateur de transition, se basant uniquement sur le contexte fourni.

D'autre part, l'agent est soumis à un budget B qui définit le nombre d'itérations maximum qu'il peut effectuer dans la boucle orange. Si ce nombre est atteint, l'agent entre alors dans un état de pré-finalisation qui lui sert à résumer les actions menées, les résultats obtenus et, le cas échéant, à expliquer la cause de l'erreur. En effet, ce budget peut être atteint car l'agent décide de continuer à utiliser des outils sans qu'il y ait d'erreur. Cet état n'est pas affiché dans la figure 1 pour garder une visualisation épurée.

Différemment de l'implémentation originale (Yao *et al.*, 2023b) qui augmentait l'espace d'action $\mathcal{A}$ du LLM avec un un espace de langage $\mathcal{L}$ pour lui permettre d'alterner librement pensées et actions, nous plaçons ici l'agent dans un cadre prédéfini. En effet, il n'a soit pas d'espace de langage (ToolChain), soit un espace dédié à la pensée qu'il ne peut pas éviter (bulle Analyste de ThinkThenAct). Cela réduit l'autonomie du processus, diminuant le nombre de décisions que doit prendre l'agent. Par cette spécificité, nous nous éloignons d'autres agents dans ce domaine (Grosnit *et al.*, 2024) qui laisse le

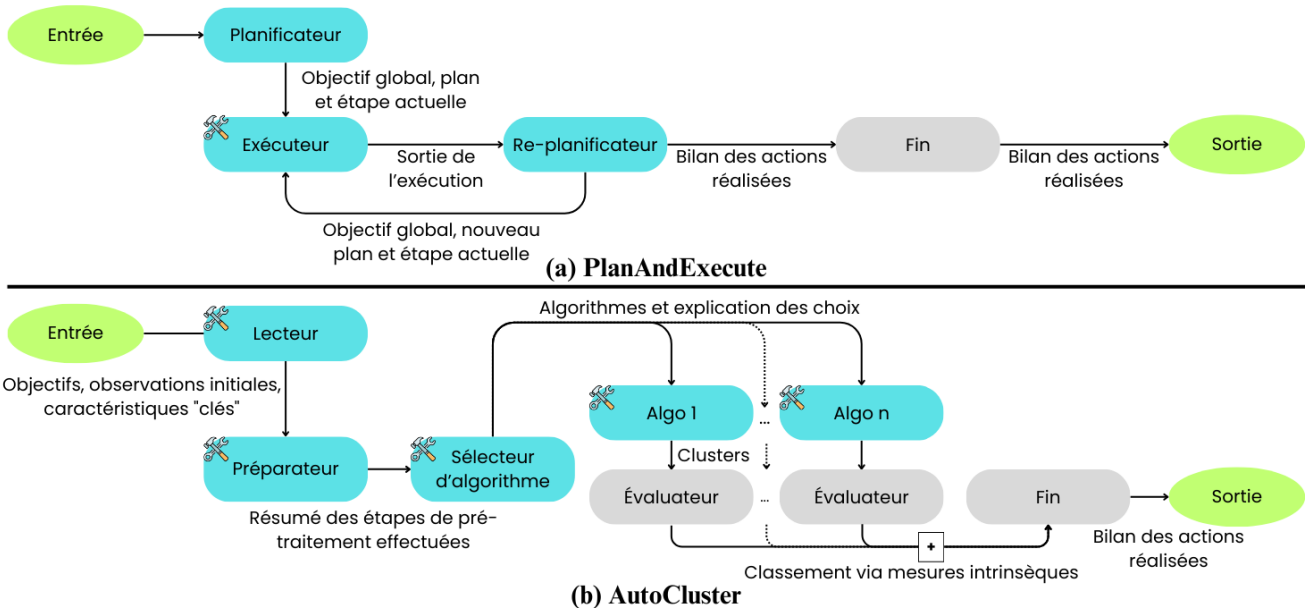


FIGURE 2 – Comparaison de PlanAndExecute et AutoCluster (**notre architecture**). En bleu les nœuds requérant un appel à un modèle de langue. Le logo outil indique qu'à cette étape, l'agent peut faire appel à un outil.

LLM choisir ses actions intrinsèques (comme prendre la décision ou non d'appeler l'analyste). Nous nous rapprochons plutôt d'un autre pan de la littérature récente (Xia *et al.*, 2024; Bouzenia *et al.*, 2024). La comparaison des deux architectures va nous permettre de déterminer les points positifs et négatifs d'avoir une étape de langage supplémentaire (bulle Analyste de ThinkThenAct) avant d'effectuer une action.

3.2.2 PlanAndExecute Clusterer

Cette seconde architecture, inspirée de Plan-And-Solve (Wang *et al.*, 2023b) (implémentation basée sur Langgraph [3]) est une version plus élaborée de ReAct, basée sur la capacité de planification des LLMs, qui implique deux étapes principales. L'étape de *planification* permet au modèle d'organiser et d'anticiper son approche en découpant la tâche primaire en sous-tâches potentiellement plus simples à exécuter. À travers l'espace de pensée explicitement donné au modèle pour générer son plan, on réduit notamment les risques de manquer une étape intermédiaire. L'étape de *résolution* implique notamment l'utilisation d'outils ou bien simplement la génération de pensées intermédiaires pour solutionner chaque sous-étape une à une.

Compte tenu de la forte part d'aléatoire dans le plan généré et donc dans la résolubilité de chaque sous-étape, une étape de re-planification est insérée après chaque étape de sous-résolution. Cela permet à l'agent d'ajuster le plan en fonction de l'évolution du processus (succès ou échec d'exécution des sous-tâches, résolution de plusieurs sous-tâches en une seule, etc.), et donne également l'occasion à l'agent de terminer le processus s'il considère que l'objectif est atteint. Cette approche ne guide que très génériquement l'agent, qui reste "maître" du plan qu'il génère, ainsi que du choix de terminer ou non le processus.

3. https://langchain-ai.github.io/langgraph/tutorials/plan-and-execute/plan-and-execute/

3.2.3 AutoCluster

Pour rejoindre la littérature récente sur des questions de capacité de raisonnement et d'autonomie des agents (Xia *et al.*, 2024; Kambhampati *et al.*, 2024), nous développons une troisième architecture, dans laquelle l'agent est placé dans une machine à états finis. Plus précisément, il va suivre des étapes classiques de data science, pour lesquelles il sera précisément instruit sur les objectifs et actions à mener. L'idée est analogue au paradigme de planification-puis-résolution, sauf que l'agent n'a plus à effectuer de choix sur la marche à suivre. La solution perd en généricité (et en autonomie), mais on s'attend à avoir une meilleure performance sur la tâche finale étant donné que l'architecture est construite en fonction de celle-ci. L'architecture est décomposée en cinq sous-étapes principales ; à chaque nouvelle étape, l'agent à accès à un résumé des étapes précédentes :

- **Le lecteur** lit les données du problème et fournit une première analyse
- **Le préparateur** qui est spécifiquement instruit pour effectuer le pré-traitement des données si nécessaire (nettoyage, standardisation, encodage des variables catégorielles, etc.)
- **Le sélecteur d'algorithme** qui va choisir jusqu'à trois algorithmes de clustering pertinents à partir de ses connaissances paramétriques et de l'analyse menée
- **Les algorithmes de clustering** (bulles "Algo i") vont chacun appliquer un de ces algorithmes séparément, en deux étapes :
  - Choix des hyper-paramètres (via une recherche en grille)
  - Application de l'algorithme et obtention des résultats
- **Les évaluateurs** qui calculent des mesures intrinsèques de compacité (distances inter, intra clusters, etc.), chacun sur l'une des différentes solutions obtenues, afin d'élire un gagnant. Les mesures calculées sont : le score Silhouette, l'indice Davies-Bouldin, l'indice Calinski-Harabasz, l'indice Dunn et la somme des carrés intra-ensemble (WCSS) (Solimun & Fernades, 2022).

À la fin de cette chaîne de traitement, un algorithme vainqueur est élu à partir des mesures intrinsèques. Dans nos premières expériences, nous comparons chaque solutions générées à la solution de référence. Cela permet d'obtenir le vrai algorithme vainqueur, car les mesures intrinsèques ne s'alignent pas nécessairement avec la solution idéale. Bien que rendant le processus moins automatique, nous justifions ce choix par l'apport supplémentaire de notre agent par rapport à celles comparées. Cet agent agit comme un assistant performant pour l'utilisateur, capable d'exécuter plusieurs algorithmes, de les comparer et de présenter ses résultats sous forme de rapport (cf. ci-dessous).

Chaque étape est dotée du prompt initial de l'utilisateur décrivant l'objectif du processus T et les métadonnées nécessaires à son déroulement M_T et, le cas échéant, d'un historique comprenant des résumés des étapes précédentes H. L'agent rentre ensuite dans un environnement E comprenant un espace d'action et un espace de langage qui lui permet d'atteindre son sous-objectif actuel. En pratique, E correspond à l'un des sous-graphes présenté dans la section 3.2.1. On a donc, après chaque étape bleue avec outil des figures 2 (a) et (b) une sortie obtenue à partir de l'équation 1, dans laquelle H est vide à la première étape puis, à l'étape i se compose de résumés $r_0 \dots r_{i-1}$ formulés à chaque étape précédente. Ces résumés forment la partie immuable de H, toujours à son début, tandis que par la suite les autres éléments de H varient selon les équations 1 et 2.

Les résumés intermédiaires de chaque phase formant le début de H sont construits différemment en fonction de l'étape. La manière la plus simple de construire ce résumé est de prendre directement la sortie de l'équation 1 qui fournit un bilan des actions réalisées (transition finale figure 1). Cependant, un opérateur final de structuration est parfois appliqué.

Pour le choix des hyper-paramètres, l'agent émet une sortie **structurée** à partir d'un espace de langage

réduit $\ell \in \mathcal{L}$ qui présente les meilleurs hyper-paramètres trouvés sous forme d'une liste de paires {nom : valeur}, ce qui permet d'extraire efficacement les informations utiles du processus mené, et de ne pas submerger le contexte de l'agent avec des détails inutiles pour la suite. Le lecteur et le sélecteur d'algorithme utilisent le même procédé. Le premier fournit un rapport structuré contenant un résumé du problème, les objectifs à atteindre, des observations initiales sur le jeu de données et quelques caractéristiques clés des données ; le second génère une liste structurée de paires {algorithme : justification de la pertinence de cet algorithme}.

En revanche, la sortie du préparateur demeure un texte libre résumant les étapes menées ; étant donné que chaque pré-traitement est personnalisé en fonction de l'objectif, il est difficile de définir un format commun de sortie. Enfin, le regroupeur n'a pas besoin de générer de résumé car après cette étape il n'y a plus d'appel à un LLM (de même pour l'évaluateur). Néanmoins, un résumé en texte libre des actions menés et des échecs ou succès rencontrés est extrait du regroupeur (généré par un LLM) afin d'élaborer un rapport de toutes les actions effectuées par l'agent (du préparateur à l'évaluateur), rendant le processus plus explicable. Un tel rapport ne peut pas être généré avec les architectures précédentes, appuyant un autre avantage de notre approche.

D'autre part, nous notons qu'à aucun moment du processus décrit, l'agent doit formuler un plan ou une auto-critique (c'est-à-dire basée sur une réflexion-propre de l'agent). Ce choix volontaire permet de bénéficier des capacités des LLMs pour formuler des solutions potentielles (Kambhampati *et al.*, 2024) plutôt que pour générer des plans. Les retours sur l'exécution proviennent exclusivement des outils utilisés, et sont donc complètement extérieurs à l'agent (aucun outil ne faisant appel à un LLM, cf. ci-dessous).

3.3 Module d'action

La motivation originale de cet article réside dans l'application et la comparaison de modèles de langue sur une tâche de clustering. C'est pourquoi, quelque soit l'architecture, l'agent n'a pas à sa disposition de scripts pré-écrits mais doit générer et exécuter son propre code. L'intérêt d'un tel agent réside notamment dans son autonomie à exécuter ce genre de tâches ; plus son utilisation requiert de lui fournir des outils spécialisés, moins il est adaptable à d'autres situations. Chacun des agents testés a accès aux mêmes données en entrée, et a à sa disposition les mêmes outils.

À l'inverse d'autres approches (Li *et al.*, 2024; Qi & Wang, 2024) dans la littérature qui fournissent des "librairies" d'outils aux agents — qui sont souvent des enveloppes pour des librairies classiques (scikit-learn, scipy, pandas, numpy, etc.) — nous faisons le choix de ne rien fournir de plus à l'agent qu'un simple interpréteur Python 3.11. Celui-ci retourne la sortie standard en cas de succès, et un résumé contenant le message d'exception, la ligne et la fonction concernée en cas d'erreur. Nous postulons que les modèles de langue exploiteront plus efficacement les bibliothèques Python couramment utilisées en fouille de données, grâce à leur mémoire paramétrique acquise lors du pré-entraînement. Cette hypothèse est notamment étayée par la diminution des performances observée dans (Li *et al.*, 2024) lorsque l'agent doit utiliser la surcouche d'outil pour l'ingénierie des caractéristiques développée par les auteurs. Outre la simplicité d'implémentation, cela réduit également la taille de contexte en entrée car il n'y a plus qu'un seul outil à proposer au modèle.

Néanmoins, dû à la large part d'aléatoire dans le code généré, nous devons tenir compte de la possibilité d'erreur dans l'exécution du code, et permettre à l'agent de corriger ces erreurs. Par conséquent, chaque appel à l'outil d'exécution de code est encapsulé dans une architecture ReAct (figure 1, explications section 3.2.1). À chaque étape de sélection, le modèle est spécifiquement

prompté pour corriger le code, si erreur, en réécrivant entièrement son code (c'est la sous-tâche t_i de l'équation 2), ou bien pour choisir entre continuer à exécuter du code (écrit à la suite du précédent) et terminer le processus sinon. L'une ou l'autre architecture ReAct peut être choisie et "branchée" dans le graphe principale, comme sous-graphe d'exécution du code (bulles bleues des figures 2 a et b). Cela permet par la même occasion la comparaison de deux compromis : une consommation réduite de tokens au prix d'une plus grande variabilité (ToolChain, un appel de moins à un LLM), ou, à l'inverse, une génération potentiellement plus stable mais plus coûteuse (cf. 4.3) en tokens (ThinkThenAct).

Sur un autre sujet, nous suivons (Grosnit *et al.*, 2024; Li *et al.*, 2024) en appliquant des tests unitaires sur le code généré, afin d'augmenter la robustesse de nos approches. Ces tests génériques vont permettre de garantir des pré-requis sur les données finales. Ils vérifient notamment que le nombre de données n'a pas changé, ni le nombre de caractéristiques. Cependant, le code généré par les agents des sections 3.2.1 et 3.2.2 suit les directives du LLM, et nous ne pouvons deviner à quel endroit insérer ces tests. En revanche, avec AutoCluster nous connaissons les étapes susceptibles de modifier les données (i.e. après le préparateur). Cette particularité augmente la robustesse de notre approche, révélant un nouvel avantage de connaître à l'avance le chemin emprunté par l'agent.

Dans la même idée, une série de tests de conformité est ajoutée, pour chaque architecture, au moment de la génération du fichier CSV final contenant les attributions de clusters. Cette étape est à part, et connue pour chacune des architectures (cf. 4.1). Ces tests vont notamment vérifier la création d'une colonne supplémentaire pour l'attribution des clusters et l'absence de valeurs nulles. En cas d'échec, le LLM entre dans une boucle où il est prompté pour réessayer de sauvegarder les données au bon format, avec des informations sur la cause de l'erreur à l'étape précédente.

4 Expérimentations

4.1 Configuration

Chaque expérience consiste à donner le chemin du fichier contenant le jeu de données au modèle et à le prompter pour qu'il applique des méthodes de clustering. Pour l'architecture AutoCluster, nous n'avons pas besoin de cette deuxième précision car les sous-objectifs sont incorporés dans les prompts spécialisés à chaque étape, mais nous la laissons dans un souci de conditions identiques entre différentes approches. Pour permettre la comparaison entre les clusters générés et ceux de référence, après chaque nœud Fin, nous ajoutons un nœud qui a pour mission de sauvegarder les données regroupées par l'agent au format CSV. Pour cela, nous faisons encore une fois appel à un modèle de langue auquel nous fournissons un outil 'save_to_csv' qui s'occupe de la sauvegarde, pour peu que le modèle soit capable de fournir le nom de la structure, dans le code, contenant les données au bon format (et si elle n'existe pas, de la créer).

Ce nœud est identique pour chaque architecture, et permet l'exécution du processus complet de manière automatique (sans aucune intervention humaine). Comme nous maîtrisons complètement le flot d'AutoCluster, nous l'appelons à chaque modification des données, évitant ainsi de répéter les traitements précédents. Cela réduit le temps d'exécution et la consommation de CPU des étapes suivantes. Par exemple, après le prétraitement, les données sont sauvegardées, et leur chemin est mis à jour pour être réutilisé par l'agent. Par ailleurs, cela augmente par la même occasion l'explicabilité du système, en créant des "points de contrôles" intermédiaires des actions menées par l'agent.

L'implémentation des agents a été faite avec LangChain et LangGraph, les paramètres par défaut des

modèles ont été utilisés (température à 0,7), et le budget B a été fixé à quatre itérations maximum. Pour ces expériences, des modèles de différentes natures d'au moins 70B de paramètres ont été utilisés, avec des capacités d'appel à des outils (cf. section 3.3). Des modèles plus petits ont été testé mais leurs interactions avec les outils étaient trop aléatoires, ne permettant pas de les appliquer fiablement dans ce cadre. En parallèle, dans notre cas, nous privilégions des modèles avec une fenêtre de contexte étendue, car cela réduit le risque de dépassement. Il est difficile de prédire efficacement la taille que vont prendre les données fournies aux LLMs ; celles-ci contiennent des historiques des tours précédents, incluant notamment des réponses générées et des retours d'outils. Le contexte peut ainsi rapidement grossir en cas d'erreurs en cascade (section 3.2.1). Dans cet article, nous comparons LLaMA3.3-70B, dit versatile de par ses capacités multilingues, et sa large fenêtre de contexte lui permettant de traiter de longues entrées, DeepSeek-R1-Distill-LLaMA-70B (DS-R1-DL-70B) (DeepSeek-AI, 2025), dont la particularité est d'émettre un "raisonnement" avant de formuler sa réponse, Mistral-Large, modèle de plus grosse capacité (123B de paramètres) ; ces trois modèles ont une taille de contexte en entrée de 128 mille tokens, qui sera suffisante pour nos expérimentations. Enfin, nous nous comparons également à un modèle de source fermée, à savoir Gemini 2.0 Flash-lite — Gemini 2.0 FL (nombre de paramètres non inconnu) — qui possède une fenêtre de contexte en entrée de un million de tokens.

4.2 Métriques

Grâce à cette optimisation du processus, nous pouvons comparer effectivement les différentes approches via plusieurs métriques. Tout d'abord, des mesures génériques, indépendantes de la tâche sous-jacente peuvent informer sur des notions de coût liés aux expériences. Nous relevons donc le **temps d'exécution** (en seconde) et le **nombre de tokens total** émis par le modèle pour chaque expérience. Ces aspects peuvent permettre de relativiser des métriques plus axées sur le succès métier liée à la tâche, introduisant un compromis entre frugalité et performance.

Dans les pas de (Hong *et al.*, 2024) et (Li *et al.*, 2024) nous attribuons une note globale **CS** (pour Comprehensive Score) basée sur une moyenne des scores de performance normalisés **ANPS** (pour Average Normalized Performance Score) et sur le taux de génération d'un fichier valide **VS** (pour Valide Submission, terme emprunté aux compétitions Kaggle).

Le taux de génération valide **VS** mesure le nombre de fois que l'agent a réussi à générer un fichier en sortie, et que ce fichier est valide au regard du nombre de colonnes et de lignes de la référence. Nous indiquons également le taux de soumission **S**, qui correspond au nombre de fois où l'agent a généré un fichier de sortie, correct ou incorrect.

Le score de performance normalisé **NPS** (Normalized Performance Score) attribue un score compris entre 0 et 1, où 1 est le meilleur score, et est défini en fonction des métriques de l'expérience. Pour une métrique de type "le plus haut est le mieux", le NPS est simplement égal à cette métrique mise à l'échelle entre 0 et 1. Le but du NPS est d'avoir une mesure unifiée de la performance de l'agent indépendamment de la tâche et de sa métrique (Hong *et al.*, 2024). Dans nos expériences, nous implémentons NPS à l'aide de deux métriques classiques qui mesurent le succès de la tâche de clustering, calculées par rapport aux données de référence : l'Indice de Rand Ajusté (ARI), légèrement en faveur de solutions comportant des clusters de taille équilibrée, et l'Information mutuelle Ajustée (AMI), qui tend à favoriser des solutions avec des clusters de tailles variées. Ces deux indices sont ajustés afin de corriger l'effet du regroupement aléatoire des points et de garantir une évaluation plus robuste de la qualité du clustering. Lorsque l'agent ne parvient pas à générer un fichier valide pour comparaison avec la référence, nous attribuons automatiquement la note zéro à ces deux indices, qui

équivaut à une correspondance aléatoire entre le clustering de référence et expérimental.

Nous les mettons à l'échelle entre 0 et 1. Notre ARI est défini sur $[-0.5; 1]$, et AMI sur $]-\infty; 1]$; dans les deux cas, toutes les valeurs négatives sont arrondies à 0. En effet, pour ces deux indices, une valeur de 0 correspond à un clustering aléatoire (c'est-à-dire une correspondance fortuite entre la partition de référence et la partition expérimentale), tandis qu'une valeur de 1 indique un accord parfait. Une valeur négative reflète un clustering discordant, ce qui traduit une mauvaise qualité de regroupement. Ainsi, en ramenant ces valeurs négatives à zéro, on harmonise l'échelle des deux indices sans introduire de biais en faveur de l'un d'eux, en ne prenant en compte que les niveaux d'accord positifs.

Pour obtenir le NPS, nous devons combiner l'AMI et l'ARI sans pouvoir préjuger de la taille ou de la forme de clusters. De telle sorte que l'un des indices puisse compenser l'autre, nous formulons le NPS afin de le maximiser si soit l'ARI, soit l'AMI est haut (soit les deux). Ainsi, nous avons simplement :

$$NPS = \max(ARI_{scaled}, AMI_{scaled}) \tag{3}$$

En raison de la disponibilité limitée des ressources, nous n'avons pas pu répéter les expériences plusieurs fois. Cependant, comme la nature de la tâche reste identique d'un jeu de données à l'autre, nous considérons que l'agent a été testé à 26 reprises sur la tâche de clustering. Ainsi, nous agrégeons le NPS sur l'ensemble des jeux de données par une moyenne arithmétique, afin d'obtenir l'**ANPS**. Cela nous permet de calculer le **CS**, d'après (Hong *et al.*, 2024) :

$$\text{CS} = 0.5 \times \text{VS} + 0.5 \times \text{ANPS} \tag{4}$$

4.3 Résultats

AutoKaggle (Li *et al.*, 2024) et Agent K v1.0 (Grosnit *et al.*, 2024) n'ont pas pu être testés, soit car le code n'est pas encore open source, soit parce que l'agent est spécifiquement conçu pour fonctionner avec un LLM d'OpenAI, que nous n'avons pas évalué dans cette étude.

Remarques générales : Tout d'abord, les meilleures métriques de performance (CS, NPS, VS) sont atteintes par AutoCluster pour chacun des LLMs, sauf pour DS-R1-DL-70B qui atteint son meilleur CS avec ReAct (TC), bien que son meilleur NPS soit atteint avec AutoCluster (TC). ReAct est de manière consistante l'architecture la plus rapide et moins coûteuse en tokens, et se retrouve deuxième meilleure dans la partie moyenne, derrière AutoCluster (TC), ce qui en fait un bon compromis. En effet, AutoCluster (TTA), bien que première en performance (meilleur CS) pour Gemini et LLaMA3.3-70B, souffre d'un score très faible avec DS-R1-DL-70B et Mistral-Large, pour des raisons que nous analysons ci-après.

Remarques sur ThinkThenAct : Bien qu'affichant la meilleure et la deuxième performance toutes architectures et LLMs confondus (CS de LLaMA3.3-70B et Gemini), AutoCluster avec ThinkThenAct arrive dernière ex-æquo des architectures en CS (dans la partie Moyenne). Pour rappel, ThinkThenAct laisse un espace de langage supplémentaire au LLM avant d'appeler un outil, dans lequel il émet ses pensées pour guider l'appel à l'outil (bulle Analyste figure 1 b). Mistral-large suppose fréquemment que le message émis par l'analyste correspond à des étapes déjà menées plutôt qu'à un objectif à atteindre. Ce comportement provient du fait que l'analyste Mistral de TTA fournit souvent du code dans sa réponse et hallucine sur les sorties de son code non exécuté, ce qui conduit le sélecteur d'outil à l'un de deux scénarios :

— Il comprend qu'un code existe déjà. Celui-ci appelle donc l'interpréteur Python en lui four-

LLM	Architecture	CS	ANPS	VS | S	# tokens	Temps (sec)
LLaMA3.3-70B	AutoCluster (TC)	0.74	0.56	0.92 | 0.96	143 029	133.71
	AutoCluster (TTA)	**0.81**	0.70	0.92 | 1.00	118 067	91.15
	Plan-and-Ex. (TC)	0.6	0.43	0.77 | 0.92	40 117	44.46
	ReAct (TC)	0.61	0.45	0.77 | 0.88	11 400	**17.57**
	DataInterpreter	0.2	0.09	0.31 | 0.38	12 047	18
DS-R1-DL-70B	AutoCluster (TC)	0.61	0.52	0.69 | 0.96	128 254	225.84
	AutoCluster (TTA)	0.46	0.3	0.62 | 0.81	134 140	238.43
	Plan-and-Ex. (TC)	0.5	0.34	0.65 | 0.73	20 873	45
	ReAct (TC)	0.71	0.45	**0.96** | 0.96	13 066	26.2
Mistral-Large	AutoCluster (TC)	0.8	**0.72**	0.88 | 1.00	127 065	288.15
	AutoCluster (TTA)	0.38	0.29	0.46 | 0.54	111 966	212.36
	Plan-and-Ex. (TC)	0.67	0.46	0.88 | 0.96	98 416	145.92
	ReAct (TC)	0.7	0.52	0.88 | 0.92	**5 802**	29.04
	DataInterpreter	0	0	0 | 0	-	60.35
Gemini 2.0 FL	AutoCluster (TC)	0.7	0.58	0.81 | 1.00	94 184	151.48
	AutoCluster (TTA)	0.8	0.68	0.92 | 1.00	55 621	91.12
	Plan-and-Ex. (TC)	0.66	0.44	0.88 | 0.92	155 034	67.42
	ReAct (TC)	0.58	0.58	0.58 | 0.88	13 832	33.57
	DataInterpreter	0.72	0.52	0.92 | 0.96	-	27.15
Moyenne	**AutoCluster (TC)**	**0.71**	**0.6**	**0.83 | 0.98**	123 133	199.8
	AutoCluster (TTA)	0.61	0.49	0.73 | 0.84	104 949	158.27
	Plan-and-Ex. (TC)	0.61	0.42	0.8 | 0.87	78 610	75.7
	ReAct (TC)	0.65	0.5	0.8 | 0.91	**11 025**	**26.6**

TABLE 1 – Comparaison des LLMs et des architectures sur la tâche de clustering. TC : ToolChain ; TTA : ThinkThenAct (section 3.2.1) ; ANPS : Average Normalized Performance Score ; CS : Comprehensive Score. En gras : meilleure valeur toutes expériences confondues ; souligné : meilleure valeur par LLM.

nissant un nouveau code qu'il suppose écrit à la suite du précédent. Ce genre de scénarios déclenche des erreurs en cascade, conduisant l'agent à atteindre le budget sans obtenir de sortie valide.

— Il comprend que l'étape a déjà été effectuée, et fournit donc un résumé en sortie afin de transitionner vers Fin plutôt que d'aller vers l'exécuteur.

L'analyste est instruit pour générer une étape concise que le sélecteur doit suivre en appelant l'interpréteur Python avec du code adéquatement généré, comme le montre la figure 3. Pour comparaison, les LLMs LLaMA et Gemini savent bien mieux tirer parti de cet espace de langage (table 4.3). Pour LLaMA, nous supposons que son nombre de paramètres plus restreint l'empêche de sur-raisonner, le modèle étant plus apte à suivre simplement les instructions données (également à l'inverse de DeepSeek qui émet un long raisonnement avant de formuler sa réponse).

Remarques sur Plan-And-Execute : Nous notons également la haute variance du nombre de tokens de l'architecte Plan-And-Execute, qui, au passage, obtient des scores relativement faible, notamment par rapport à ReAct, malgré sa plus haute complexité. En l'occurrence, certains LLMs n'arrivent pas à tirer profit de cette architecture et restent bloqués longtemps une boucle, dû au re-planificateur qui décide constamment de rajouter une étape au plan, augmentant considérablement le contexte (qui contient un résumé de toutes les étapes parcourues),

Remarques sur DataInterpreter : Les scores obtenus par *DataInterpreter* sont hautement différents en fonction du LLM. Les résultats avec DS-R1-DL-70B sont absents car l'agent est constamment en erreur avec ce LLM (erreur de structuration des réponses cf. 5) ; avec Mistral, bien que ne générant pas d'erreur, le modèle est incapable de lire le fichier au chemin fournit, hallucinant invariablement

QUICKLY review the output of the previous step and assess whether the given task: "{{task}}" is already completed based on the current state of the process.
If and ONLY IF the task is not yet completed, suggest a **QUICK AND CONCISE UNIQUE** next step to move closer to the goal of given task.
Do not propose additional steps if the task is already solved, but write a prompt to stop the process and stop calling tools. Attention: **KEEP YOUR RESPONSE STRICTLY LIMITED TO THE SCOPE OF THE TASK**

FIGURE 3 – Prompt de l'Analyste the ThinkThenAct. En bleu une variable remplacée par le contexte (cf. 3.2.1)

un chemin différent de celui fournit. En fin de compte, seul Gemini parvient à performer correctement dans ce cadre, obtenant le deuxième meilleur résultat après TTA.

Remarques sur la frugalité des approches : Nos agents sont fréquemment plus coûteux en temps et en tokens que les architectures classiques. Notamment à cet égard, ReAct apparaît comme une solution viable moins coûteuse. Nous soulignons néanmoins la plus grande diversité des solutions explorées par nos agents, capables de produire un rapport plus exhaustif sur les approches testées. Par ailleurs, nous supposons que l'écart de performance entre AutoCluster et ReAct se creusera davantage sur des tâches plus complexes nécessitant plus de traitements.

Remarques finales sur les LLMs : Bien que théoriquement agnostique au modèle, nous notons d'importantes différences de performance en fonction du LLM utilisé par l'agent. Comme noté ci-dessus, les modèles ne réagissent pas de la même manière en fonction des prompts et de l'architecture, démontrant la nécessité de **coupler** un minimum le design de l'agent au LLM. Il faut notamment prendre en compte la taille du LLM et sa capacité à raisonner. Les modèles fermés, comme Gemini, rendent ce travail plus complexe, et limite l'interprétation des résultats. Nous ne pouvons pas analyser en profondeur les raisons de la relative bonne performance de Gemini n'ayant que très peu d'informations sur ce modèle.

5 Discussion sur les difficultés rencontrées

Sans que cela ne soit clairement explicité dans la littérature, les différentes solutions que nous avons mises en place nécessitent que les modèles vérifient un certain nombre de pré-requis. Tout d'abord, le modèle doit être capable de générer une sortie *structurée*, pour permettre notamment l'appel à un outil (Schick *et al.*, 2023) ou tout simplement pour pouvoir structurer sa génération en code. Sans cette capacité, les implémentations proposées ne sont pas applicables car le flot de travail est basé sur ce parsing (en particulier pour les architectures plus "autonomes"). Or, cette fonctionnalité n'est généralement pas apprise lors du pré-entraînement et nécessite une étape de fine-tuning [4], ce qui réduit l'éventail des modèles branchables sur notre architecture. De plus, dû à la complexité de la tâche (Shen *et al.*, 2024), qui requiert une compréhension de l'outil à disposition, savoir quand l'invoquer (donc des capacités de planification), comment, et comprendre les résultats, les "petits" modèles peinent à l'accomplir. Les modèles prêts à l'emploi (sans nécessité de fine-tuning spécifique à nos outils) dotés de cette capacité génériquement sont généralement assez gros (autour de 50 - 70 milliards de paramètres). De plus, même apprise, cette fonctionnalité n'est pas totalement fiable, il n'est pas rare de voir les modèles échouer à générer des formats complexes (par exemple lorsque le format cible contient une *union* de type "répond en tel format *ou* en tel format"). Des variantes de ce problème existent, le rendant plus compliqué à détecter :

4. https://platform.openai.com/docs/guides/function-calling

— La fonction est appelé dans le corps du message, et non dans l'argument dédié à cet effet
— Le nom de l'outil appelé ne correspond à aucun outil à disposition du modèle mais le format est correct
— Les arguments passés à l'outil correspondent en typage (chaîne de caractère, nombre, etc.) mais pas en sémantique

La solution apportée dans ce cadre est d'encapsuler toute génération risquée (requérant un format particulier en sortie) dans une boucle de relance qui va re-prompter le modèle en cas d'échec en insistant sur la cause de l'erreur. Ce dernier facteur est essentiel car il permet au modèle de se focaliser sur la raison de son échec, à l'inverse d'un prompt générique qui demanderait au modèle de refaire sa génération suite à une erreur.

Un deuxième point source de contraintes provient des différences inhérentes aux LLMs (différence de données de pré-entraînement, différents objectifs de fine-tuning, etc.). Bien que théoriquement agnostique au modèle, notre cadre a dû évolué au fur et à mesure que de nouveaux LLMs ont été testés pour prendre en compte les subtilités liés à chacun. Par exemple, les modèles Mistral testés impose un historique des messages bien particulier (un message d'outil doit être le dernier d'une conversation), alors que cette limitation n'existait pas avec un LLaMA ; un Gemini qui échoue à appeler un outil va générer un message vide (ce sont ses métadonnées qui indique la cause de l'erreur) or certains modèles fonctionnent mal avec les messages vides dans l'historique. Ce sont autant de problèmes qu'il a fallu traiter au cas par cas, provenant potentiellement du manque du maturité et de consensus global sur les approches à adopter pour ce genre de questions. Ce sujet demeure en effet extrêmement récent.

6 Conclusion

Dans cet article, nous expliquons AutoCluster, un agent basé sur les LLMs spécialisé pour le clustering. Nous démontrons l'avantage de connaître à l'avance le chemin suivi par l'agent, augmentant la robustesse des résultats produits (tests de conformité), et bénéficiant des capacités de leur capacité à formuler des solutions potentielles (Kambhampati *et al.*, 2024) plutôt que de reposer sur leur capacité de planification. Dans la même idée, le LLM ne s'auto-critique pas mais reçoit le feedback de l'interpréteur Python, lui permettant d'adapter son approche avec plus de sûreté. Nous démontrons la supériorité de notre approche à travers des expérimentations sur des tâches de clustering (cf. 4.3), concept à partir duquel a été bâti AutoCluster.

De nombreuses améliorations sont envisagées, principalement pour notre agent AutoCluster. Une première piste consiste à augmenter les capacités de perception de l'agent, en tirant profit des modèles multimodaux auxquels on peut passer les images générés en code directement dans le contexte. Ensuite, dans l'idée des RAG, il serait intéressant d'évaluer les performances de l'agent auquel on fournit en contexte des données récupérées des pages scikit-learn ou scipy sur les algorithmes sélectionnés (qui incluent notamment des cas d'usage et des descriptions des paramètres). Pour étendre cette direction et rejoindre la littérature (Wang *et al.*, 2023a; Grosnit *et al.*, 2024; Hong *et al.*, 2024), nous allons concentrer nos travaux sur l'implémentation d'une mémoire à long terme, avec requêtage actif à la manière de (Grosnit *et al.*, 2024). Enfin, nous souhaitons complexifier la tâche donnée à l'agent, avec des données moins structurées et des clusters moins bien définis, afin d'évaluer plus profondément les avantages de notre approche. Dans la même idée, il serait intéressant d'étendre le périmètre de l'agent à d'autres tâches d'analyse de données comme la fouille de motif ou la découverte de sous-groupes.

Références

BESTA M., BLACH N., KUBICEK A., GERSTENBERGER R., PODSTAWSKI M., GIANINAZZI L., GAJDA J., LEHMANN T., NIEWIADOMSKI H., NYCZYK P. *et al.* (2024). Graph of thoughts : Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, p. 17682–17690.

BOMMASANI R., HUDSON D. A., ADELI E., ALTMAN R., ARORA S., VON ARX S., BERNSTEIN M. S., BOHG J., BOSSELUT A., BRUNSKILL E. *et al.* (2021). On the opportunities and risks of foundation models. *arXiv preprint arXiv :2108.07258*.

BOUZENIA I., DEVANBU P. & PRADEL M. (2024). RepairAgent : An Autonomous, LLM-Based Agent for Program Repair. arXiv :2403.17134 [cs].

BROWN T., MANN B., RYDER N., SUBBIAH M., KAPLAN J. D., DHARIWAL P., NEELAKANTAN A., SHYAM P., SASTRY G., ASKELL A. *et al.* (2020). Language models are few-shot learners. *Advances in neural information processing systems*, **33**, 1877–1901.

CHAN J. S., CHOWDHURY N., JAFFE O., AUNG J., SHERBURN D., MAYS E., STARACE G., LIU K., MAKSIN L., PATWARDHAN T., WENG L. & MĄDRY A. (2025). MLE-bench : Evaluating Machine Learning Agents on Machine Learning Engineering. arXiv :2410.07095 [cs], DOI : 10.48550/arXiv.2410.07095.

CHASE H. (2022). LangChain.

DEEPSEEK-AI (2025). Deepseek-r1 : Incentivizing reasoning capability in llms via reinforcement learning.

GROSNIT A., MARAVAL A., DORAN J., PAOLO G., THOMAS A., BEEVI R. S. H. N., GONZALEZ J., KHANDELWAL K., IACOBACCI I., BENECHEHAB A., CHERKAOUI H., EL-HILI Y. A., SHAO K., HAO J., YAO J., KEGL B., BOU-AMMAR H. & WANG J. (2024). Large Language Models Orchestrating Structured Reasoning Achieve Kaggle Grandmaster Level. arXiv :2411.03562 [cs], DOI : 10.48550/arXiv.2411.03562.

GUO S., DENG C., WEN Y., CHEN H., CHANG Y. & WANG J. (2024). DS-Agent : Automated Data Science by Empowering Large Language Models with Case-Based Reasoning. arXiv :2402.17453 [cs], DOI : 10.48550/arXiv.2402.17453.

GUU K., LEE K., TUNG Z., PASUPAT P. & CHANG M. (2020). Retrieval augmented language model pre-training. In *International conference on machine learning*, p. 3929–3938 : PMLR.

HONG S., LIN Y., LIU B., LIU B., WU B., LI D., CHEN J., ZHANG J., WANG J., ZHANG L., ZHANG L., YANG M., ZHUGE M., GUO T., ZHOU T., TAO W., WANG W., TANG X., LU X., ZHENG X., LIANG X., FEI Y., CHENG Y., XU Z. & WU C. (2024). Data Interpreter : An LLM Agent For Data Science. arXiv :2402.18679 [cs].

HU X., ZHAO Z., WEI S., CHAI Z., MA Q., WANG G., WANG X., SU J., XU J., ZHU M., CHENG Y., YUAN J., LI J., KUANG K., YANG Y., YANG H. & WU F. (2024). InfiAgent-DABench : Evaluating Agents on Data Analysis Tasks. arXiv :2401.05507 [cs], DOI : 10.48550/arXiv.2401.05507.

HUANG J., CHEN X., MISHRA S., ZHENG H. S., YU A. W., SONG X. & ZHOU D. (2024a). Large Language Models Cannot Self-Correct Reasoning Yet. arXiv :2310.01798 [cs], DOI : 10.48550/arXiv.2310.01798.

HUANG Q., VORA J., LIANG P. & LESKOVEC J. (2024b). MLAgentBench : Evaluating Language Agents on Machine Learning Experimentation. arXiv :2310.03302 [cs], DOI : 10.48550/arXiv.2310.03302.

JIANG Z., SCHMIDT D., SRIKANTH D., XU D., KAPLAN I., JACENKO D. & WU Y. (2025). AIDE : AI-Driven Exploration in the Space of Code. arXiv :2502.13138 [cs], DOI : 10.48550/arXiv.2502.13138.

JING L., HUANG Z., WANG X., YAO W., YU W., MA K., ZHANG H., DU X. & YU D. (2025). DSBench : How Far Are Data Science Agents to Becoming Data Science Experts ? arXiv :2409.07703 [cs], DOI : 10.48550/arXiv.2409.07703.

KAMBHAMPATI S., VALMEEKAM K., GUAN L., VERMA M., STECHLY K., BHAMBRI S., SALDYT L. P. & MURTHY A. B. (2024). Position : Llms can't plan, but can help planning in llm-modulo frameworks. In *Forty-first International Conference on Machine Learning*.

LEWIS P., PEREZ E., PIKTUS A., PETRONI F., KARPUKHIN V., GOYAL N., KÜTTLER H., LEWIS M., YIH W.-T., ROCKTÄSCHEL T., RIEDEL S. & KIELA D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, volume 33, p. 9459–9474 : Curran Associates, Inc.

LI Z., ZANG Q., MA D., GUO J., ZHENG T., LIU M., NIU X., WANG Y., YANG J., LIU J., ZHONG W., ZHOU W., HUANG W. & ZHANG G. (2024). AutoKaggle : A Multi-Agent Framework for Autonomous Data Science Competitions. arXiv :2410.20424, DOI : 10.48550/arXiv.2410.20424.

LIU B., JIANG Y., ZHANG X., LIU Q., ZHANG S., BISWAS J. & STONE P. (2023). LLM+P : Empowering Large Language Models with Optimal Planning Proficiency. arXiv :2304.11477 [cs], DOI : 10.48550/arXiv.2304.11477.

MA J., DAI D., SHA L. & SUI Z. (2024). Large Language Models Are Unconscious of Unreasonability in Math Problems. arXiv :2403.19346 [cs], DOI : 10.48550/arXiv.2403.19346.

QI D. & WANG J. (2024). CleanAgent : Automating Data Standardization with LLM-based Agents. arXiv :2403.08291 [cs].

REMIL Y., BENDIMERAD A., MATHONAT R., CHALEAT P. & KAYTOUE M. (2021). " what makes my queries slow ?" : Subgroup discovery for sql workload analysis. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, p. 642–652 : IEEE.

SCHICK T., DWIVEDI-YU J., DESSÌ R., RAILEANU R., LOMELI M., HAMBRO E., ZETTLEMOYER L., CANCEDDA N. & SCIALOM T. (2023). Toolformer : Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, **36**, 68539–68551.

SHEN W., LI C., CHEN H., YAN M., QUAN X., CHEN H., ZHANG J. & HUANG F. (2024). Small LLMs Are Weak Tool Learners : A Multi-LLM Agent. arXiv :2401.07324 [cs], DOI : 10.48550/arXiv.2401.07324.

SHEN Y., SONG K., TAN X., LI D., LU W. & ZHUANG Y. (2023). HuggingGPT : Solving AI Tasks with ChatGPT and its Friends in Hugging Face. arXiv :2303.17580 [cs], DOI : 10.48550/arXiv.2303.17580.

SHINN N., CASSANO F., GOPINATH A., NARASIMHAN K. & YAO S. (2023). Reflexion : Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, **36**, 8634–8652.

SOLIMUN S. & FERNADES A. A. R. (2022). Cluster analysis on various cluster validity indexes with average linkage method and euclidean distance (study on compliant paying behavior of bank x customers in indonesia 2021). *Mathematics and Statistics*, **10**(4), 747–753.

VISWANATHAN V., GASHTEOVSKI K., GASHTEOVSKI K., LAWRENCE C., WU T. & NEUBIG G. (2024). Large language models enable few-shot clustering. *Transactions of the Association for Computational Linguistics*, **12**, 321–333.

WANG G., XIE Y., JIANG Y., MANDLEKAR A., XIAO C., ZHU Y., FAN L. & ANANDKUMAR A. (2023a). Voyager : An Open-Ended Embodied Agent with Large Language Models. arXiv :2305.16291 [cs].

WANG L., XU W., LAN Y., HU Z., LAN Y., LEE R. K.-W. & LIM E.-P. (2023b). Plan-and-Solve Prompting : Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models. arXiv :2305.04091 [cs], DOI : 10.48550/arXiv.2305.04091.

WEI J., WANG X., SCHUURMANS D., BOSMA M., XIA F., CHI E., LE Q. V., ZHOU D. *et al.* (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, **35**, 24824–24837.

XI Z., CHEN W., GUO X., HE W., DING Y., HONG B., ZHANG M., WANG J., JIN S., ZHOU E. *et al.* (2025). The rise and potential of large language model based agents : A survey. *Science China Information Sciences*, **68**(2), 121101.

XIA C. S., DENG Y., DUNN S. & ZHANG L. (2024). Agentless : Demystifying LLM-based Software Engineering Agents. arXiv :2407.01489.

YAO S., YU D., ZHAO J., SHAFRAN I., GRIFFITHS T., CAO Y. & NARASIMHAN K. (2023a). Tree of thoughts : Deliberate problem solving with large language models. *Advances in neural information processing systems*, **36**, 11809–11822.

YAO S., ZHAO J., YU D., DU N., SHAFRAN I., NARASIMHAN K. & CAO Y. (2023b). React : Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

ZHANG Y., PAN Y., WANG Y. & CAI J. (2024). PyBench : Evaluating LLM Agent on various real-world coding tasks. arXiv :2407.16732 [cs], DOI : 10.48550/arXiv.2407.16732.

ZHANG Y., WANG Z. & SHANG J. (2023). ClusterLLM : Large Language Models as a Guide for Text Clustering. arXiv :2305.14871, DOI : 10.48550/arXiv.2305.14871.