Actes de CORIA-TALN 2026
Nantes, France, 29 juin au 3 juillet 2026

Quantification du conflit entre mémoire paramétrique et mémoire contextuelle : Étude des Métriques de Dégradation sur la Famille de modèles Qwen3.5

Hugo De Bosschere, Maximilien Lyonnais, Clément Géliot, Joël Legrand

RÉSUMÉ

Le Retrieval Augmented Generation (RAG) est une technique consistant à extraire des documents depuis une base de données pour les injecter dans le contexte du modèle. Les Grands Modèles de Langue (Large Language Models ou LLM en anglais) s'appuient de plus en plus sur du RAG pour exploiter des informations récentes ou spécifiques. Cette approche est néanmoins source de conflit entre la *mémoire paramétrique* (le savoir figé dans les poids du modèle) et la *mémoire contextuelle* (les données injectées dans le prompt via le RAG). Un conflit émerge lorsque ces deux mémoires se contredisent : comment le modèle selectionne-t-il alors la source d'information à privilégier ? Nous étudions ici ce conflit via la mise à jour d'Interface de Programmation d'Application (Application Programming Interface ou API en anglais). Nous simulons une situation où un LLM, entraîné sur une version antérieure de `NumPy`, reçoit en contexte la documentation de sa nouvelle version. Nous quantifions dans cet article la propension des modèles de la famille qwen3.5 à rejeter leur mémoire paramétrique au profit de la mémoire contextuelle.

ABSTRACT

Quantifying the conflict between parametric and contextual memory: Degradation Metrics Study on the Qwen3.5 Model Family

Retrieval Augmented Generation (RAG) is a technique that retrieves documents from a database to inject them in the model's context. Large Language Models (LLM) increasingly rely on RAG to use recent or specific information. This creates a conflict between the *parametric memory* (knowledge stored in the model's weights) and the *contextual memory* (knowledge injected in the prompt through RAG). When these two sources of information clash, it creates a conflict: how will then the model choose which information to prefer ? We study here this conflict through Application Programming Interface (API) updates. We simulate a situation where an LLM trained on a previous version of `NumPy` receives in its context the documentation of the new version. We quantify in this article the tendency of qwen3.5 models to choose their contextual memory over their parametric memory.

MOTS-CLÉS : Retrieval-Augmented Generation, RAG, Grands Modèles de Langage, LLM, Mémoire Paramétrique, Mémoire Contextuelle, Ingénierie Logicielle, Évaluation.

KEYWORDS: Retrieval-Augmented Generation, RAG, Large Language Models, LLM, Parametric Memory, Contextual Memory, Software Engineering, Evaluation.

Actes de l'atelier sur l'évaluation des modèles génératifs (LLM) et challenges (EvalLLM 2026), pages 141–159.

1 Introduction

Les LLMs ont démontré des capacités remarquables, allant de la création de textes inédits à la génération de code complexe. Cependant, ces modèles souffrent de limitations inhérentes à leur conception : ils sont sujets aux hallucinations (c'est-à-dire que les textes qu'ils génèrent contiennent des contre-vérités) (Sun *et al.*, 2024) et demeurent strictement contraints par les connaissances figées dans leur corpus d'entraînement. Ils sont ainsi incapables, par nature, de répondre de manière fiable à des requêtes portant sur des informations récentes ou privées.

Pour pallier ce problème, la technique du *Retrieval-Augmented Generation* s'est imposée comme un standard (Lewis *et al.*, 2020). Le RAG consiste à extraire des documents pertinents depuis une base de données externe pour les injecter directement dans la fenêtre de contexte du modèle avant l'étape de génération.

Cette nouvelle technologie met néanmoins en lumière une tension fondamentale au cœur de l'architecture des LLMs, partagés entre deux sources de connaissances :

— **La mémoire paramétrique** : le savoir encodé de manière statique dans les poids du réseau de neurones lors de sa phase d'entraînement.

— **La mémoire contextuelle** : les informations éphémères injectées directement dans le prompt lors de l'inférence.

Un conflit majeur émerge lorsque ces deux mémoires se contredisent. Comment le modèle selectionne-t-il la source d'information à privilégier ? Longpre *et al.* (2021) ont démontré que les LLMs affichent une forte préférence pour leurs données d'entraînement, même face à des faits substitués en contexte. Xie *et al.* (2023) ont montré que cette susceptibilité dépend fortement de la cohérence du contexte. Plus récemment, Wu *et al.* (2024) avec le benchmark *ClashEval* ont proposé de quantifier ce « bras de fer » sur des tâches de questions-réponses en langage naturel. Ils ont confirmé les résultats de Xie et al. concernant l'importance de la cohérence du contexte sur le choix du LLM de se fier plutôt à sa mémoire paramétrique ou à sa mémoire contextuelle.

Notre étude prolonge ces travaux en explorant ce conflit dans un domaine concret : l'ingénierie logicielle et l'adaptation aux mises à jour d'APIs. Dans un environnement réel, les bibliothèques évoluent. Un LLM assistant de code a nécessairement été entraîné sur une version antérieure. S'il reçoit en contexte la documentation de la nouvelle version (en supposant un RAG parfait), le comportement attendu est qu'il privilégie *systématiquement* le contexte. Or, les LLMs ne sont pas entraînés à se fier exclusivement à leur contexte face à des connaissances massivement ancrées dans leurs poids, comme l'usage de la librairie `NumPy` (Harris *et al.*, 2020).

Nos contributions sont les suivantes :

— **Développement d'un cadre d'évaluation fonctionnel** par exécution stricte du code généré.

— **Introduction de métriques de dégradation** permettant d'isoler la perte de cohérence syntaxique de la perte d'intelligence.

— **Quantification des pertes de performances** des modèles de la famille qwen3.5 sur deux perturbations différentes.

2 Méthodologie et Cadre Expérimental

Notre but était donc de mesurer et d'évaluer à quel point les LLMs sont capables de s'adapter à des informations qui ne sont pas seulement nouvelles mais qui sont en conflit direct avec les données sur lesquelles ils ont été entraînés. Pour mesurer cette capacité d'adaptation et, le cas échéant, la perte de performance liée à cette nécessité d'adaptation nous avons tout d'abord généré des documentations contrefactuelles de trois types (ultra-minimale, minimale et explicative). Pour voir à quoi ressemble ces trois types de documentation, voir en annexe A.3. Ensuite nous avons fourni cette documentation au LLM via le moteur d'inférence llama.cpp (Gerganov, 2023) et le wrapper haut-niveau ollama (Ollama Contributors, 2024). Nous avons fourni au LLM la documentation directement dans sa fenêtre de contexte en s'assurant que la taille de la documentation était inférieure à la taille maximale de la fenêtre de contexte habituelle du LLM. Nous avons utilisé le benchmark DS1000 (Lai *et al.*, 2023) en tant que base de problèmes à résoudre en ne gardant que les problèmes liés à la librairie `NumPy` (ce qui fait un total de 159 problèmes à résoudre) et en modifiant les énoncés des problèmes pour s'assurer que la perturbation `NumPy` idoine y était présente aussi (pour assurer la cohérence du contexte). Ensuite nous avons récupéré le code généré par le LLM via des méthodes de parsing. Finalement, nous avons fait de l'évaluation fonctionnelle du code généré pour nous assurer de deux choses : Premièrement que le code utilise bien les modifications censées correspondre à une mise à jour d'une librairie. Deuxièmement que le code qui utilise les modifications idoines fonctionne correctement et renvoie les bonnes valeurs sur les différents test cases de DS1000. La figure 1 résume notre pipeline.

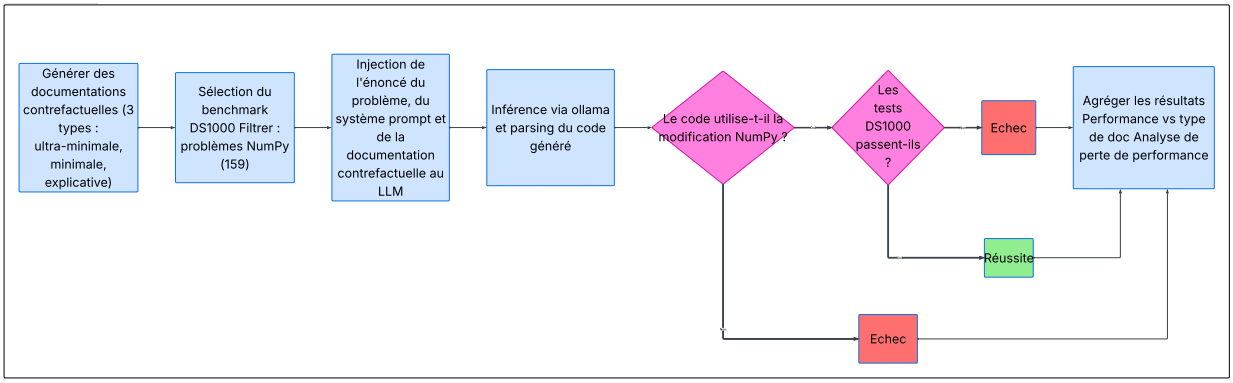


FIGURE 1 – Pipeline d'évaluation : de la conception de la documentation contrefactuelle à la double évaluation dynamique via wrappers et nettoyage AST.

2.1 Documentations Contrefactuelles et Perturbations

La génération de la documentation est déterministe. Nous évaluons trois niveaux de documentation : **Ultra-minimale** (catalogue des noms), **Minimale** (noms, arguments et signature), et **Explicative** (explicitation textuelle de la règle de perturbation).

Nous appliquons deux types de perturbations appliquées à l'attribut terminal :

— **Suffixe _v2 :** Ajout systématique de `_v2` (ex: `np.mean_v2`).

— **Capitale :** Mise en majuscule de la première lettre (ex: `np.Mean`).

2.2 Double Évaluation Fonctionnelle et Parsing AST

Contrairement aux évaluations textuelles classiques, notre pipeline valide le code par une *double exécution dynamique* :

1. **Évaluation avec injection (Wrapper) :** Le code est passé à un wrapper strict qui intercepte les accès à l'objet `np`. Si la convention perturbée n'est pas respectée, une `MissingSuffixError` est levée.
2. **Évaluation de contrôle :** Le même code, après suppression algorithmique des perturbations via nettoyage de l'Abstract Syntax Tree (AST), est exécuté avec l'import `numpy` standard.

Ainsi, le code est exécuté à la fois sur la librairie perturbée et sur la librairie originale. Cette méthode permet de vérifier si un code qui échoue à appliquer la perturbation était néanmoins algorithmiquement correct avec la version classique de `NumPy`.

2.3 Pass@10, prompt et hyperparamètres

Nous avons évalué la famille qwen3.5 (Bai *et al.*, 2023) (0.8B, 2B, 4B, 9B, 27B, 35B) en inférence locale via ollama, en utilisant la métrique probabiliste $pass@10$ (Chen *et al.*, 2021).

Le pass@k est un paradigme d'évalutation des LLM où l'on cheche à trouver la probabilité que le LLM réussise la tâche au moins une fois s'il dispose de k essais. Ce paradigme d'évaluation est plus tendre au sens où la probabilité du pass@k est forcément supérieure ou égale à celle du pass@1. Si l'on note n le nombre d'échantillons dont nous disposons pour évaluer le pass@k, nous avons fait le choix d'utiliser n = k. Il est tout à fait possible de calculer le pass@k avec un nombre d'échantillons n > k (par exemple 50 échantillons pour le pass@10) puis en calculant le nombre de combinaisons de 10 échantillons parmi lesquels il y a au moins une réussite divisé par le nombre total de combinaisons de 10 échantillons. Néanmoins, nous avons ici été contraint par la puissance de calcul à notre disposition.

Pour s'assurer de la reproductibilité de nos résulats, nous avons fixé une seed de départ pour le pass@10 (42) que nous avons incrémenté de 1 à chaque fois que le modèle se trompait et qu'il fallait donc relancer l'inférence.

Pour plus de détails sur les paramètres d'inférence des LLMs voir en annexe A.1.

Nous utilisions systématiquement le même system prompt au début de chaque inférence pour essayer de contraindre le LLM au maximum de respecter un format de réponse. L'entièreté du system prompt peut être retrouvé en annexe A.2.

3 Résultats et Analyse : la Famille Qwen3.5

Nous analysons dans cette partie les résultats obtenus lors de nos expériences.

3.1 Loi d'échelle et capacités de base

Sur les plis de contrôle (sans conflit), les performances croissent strictement avec le nombre de paramètres. Le taux de réussite passe de 15 % (0.8B) à 81 % (35B). Comme illustré sur la figure 2, cette évolution semble suivre une relation logarithmique :

$$y = a \cdot \log(p) + b \tag{1}$$

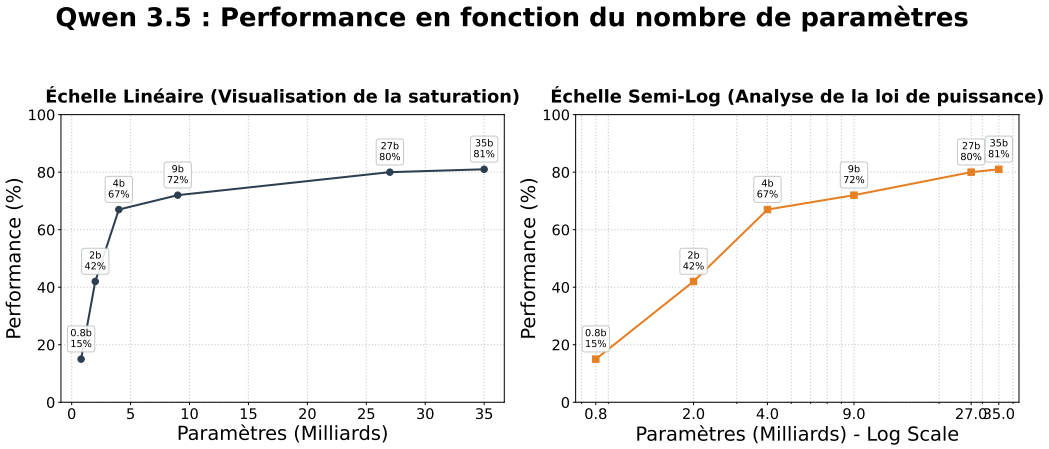


FIGURE 2 – Performance des modèles de la famille Qwen3.5 sur les plis de contrôle en fonction du nombre de paramètres (gauche) et du logarithme du nombre de paramètres (droite).

Ainsi il semble que plus un modèle comporte un nombre important de paramètres plus il est performant sur des tâches de code qui requièrent une syntaxe usuelle.

3.2 Première analyse des performances sur les plis contrefactuels

Nous allons analyser dans cette partie les résultats des évaluations résumés dans la figure 3.

Pour la fluidité des analyses que nous allons mener par la suite, il est important de noter que bien que qwen3.5:35b soit de la même famille que les autres modèles (même corpus d'entraînement, même structure de post-training) son architecture est différente des autres modèles puisqu'il se base sur une architecture Mixture-of-Experts (MoE) ce qui accélère l'inférence mais peut avoir un impact négatif sur la performance, notamment sur le raisonnement.

On constate aussi que les LLMs performent systématiquement mieux lors des plis de tests avec la documentation explicative plutôt qu'avec les documentations minimales ou ultra minimales. Tout se passe comme si le fait de devoir généraliser la perturbation provoquait une baisse de compétence du LLM. Ainsi, aider le LLM en lui expliquant la règle plutôt qu'en le laissant l'inférer sur un grand nombre d'exemples semblerait lui laisser moins de possibilités pour se tromper. C'est malheureux dans notre cas puisque nous souhaiterions que le LLM puisse réussir à utiliser seul la nouvelle documentation de la librairie.

Petite note sur l'interprétation du graphique 3 : Il pourrait arriver que la somme du pourcentage de réussite sur l'évaluation sur la librairie contrefactuelle et sur la librairie d'origine dépasse le pourcentage de réussite sur le pli de contrôle. En effet, comme l'évaluation est fonctionnelle et

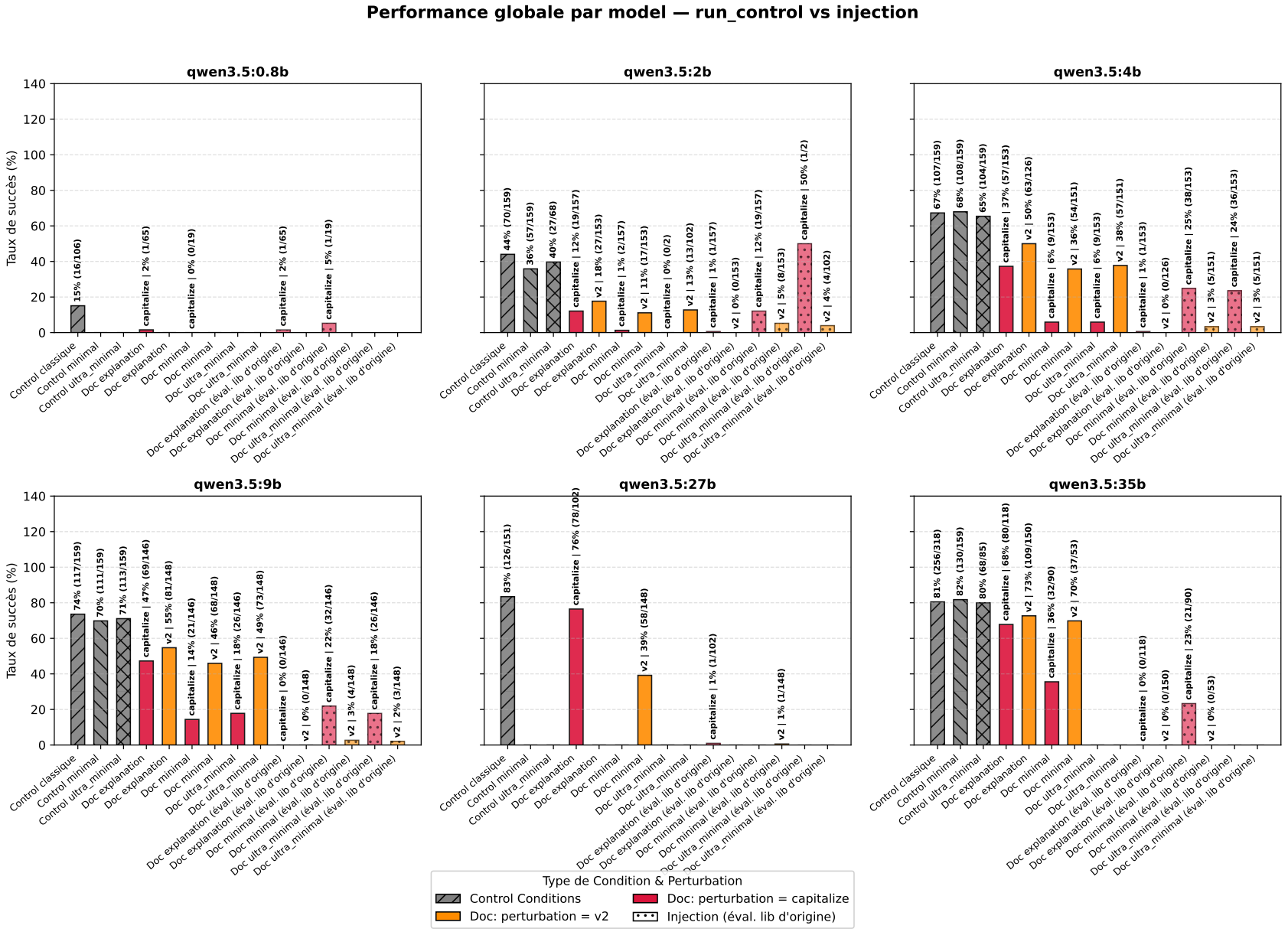


FIGURE 3 – Performances des modèles de la famille qwen3.5 sur les plis de contrôle (classique, documentation minimale et ultra minimale) et d'injections avec les perturbations _v2 et capitale (avec en contexte les documentations explicatives, minimales et ultra minimales) évalués sur la librairie d'origine (eval. lib d'origine) et la librairie contrefactuelle

que certains problèmes peuvent être résolus sans utiliser la librairie `NumPy`, il arrive que certaines solutions soient comptabilisées comme correctes à la fois pour la librairie de base et la librairie contrefactuelle. Pour éviter ce cas de figure, nous avons décidé d'exclure les fonctions qui passent les deux évaluations.

3.3 Métriques de dégradation

On introduit ici quelques définitions qui nous permettront par la suite de mener des analyses plus approfondies : Soient $\mathcal{E}_{ctrl}$ et $\mathcal{E}_{inj}$ les ensembles d'évaluations de contrôle et d'injection. On définit les sous-ensembles de succès :

$$\begin{aligned} S_{ctrl} &= \{e \in \mathcal{E}_{ctrl} \mid \text{control\_passed}(e)\} \\ S_{inj} &= \{e \in \mathcal{E}_{inj} \mid \text{injection\_passed}(e)\} \\ S_{inj\_ctrl} &= \{e \in \mathcal{E}_{inj} \mid \text{control\_passed}(e)\} \end{aligned}$$

où control_passed signifie que le code généré fonctionne quand il est exécuté avec la librairie originale, et injection_passed quand il fonctionne avec le wrapper. On remarquera que S_{ctrl_inj} n'est pas pris en compte puisque l'on considère que cet ensemble est forcément vide (pourquoi le LLM générerait-il un code perturbé alors que la perturbation ne lui a même pas été précisée ?)

1. La perte de performance L_{perf} mesure la baisse de réussite stricte sur la tâche qui nous intéresse au final :

$$L_{perf} = \frac{|S_{ctrl}|}{|\mathcal{E}_{ctrl}|} - \frac{|S_{inj}|}{|\mathcal{E}_{inj}|} \tag{2}$$

2. La perte de compétence L_{comp} sanctionne l'incapacité à produire un code valide pour au moins l'une des deux librairies :

$$L_{comp} = \frac{|S_{ctrl}|}{|\mathcal{E}_{ctrl}|} - \frac{|S_{inj} \cup S_{inj\_ctrl}|}{|\mathcal{E}_{inj}|} \tag{3}$$

3. La perte d'intelligence L_{int} s'appuie sur l'ensemble C des confusions (erreurs d'assertion) :

$$C = \{e \in \mathcal{E}_{inj} \mid e \notin (S_{inj} \cup S_{inj\_ctrl}) \wedge (\text{stdout}(e) \in \text{Err}_{assert} \vee \text{stdout\_control}(e) \in \text{Err}_{assert})\} \tag{4}$$

$$L_{int} = \frac{|C|}{|\mathcal{E}_{inj}|} \tag{5}$$

4. La perte de cohérence L_{coh} est définie comme le résidu de la perte de compétence :

$$L_{coh} = L_{comp} - L_{int} \tag{6}$$

3.4 Retour à l'analyse

On constate systématiquement (bien qu'à des degrés différents en fonction de la taille du modèle, du type de perturbation et du type de documentation) une perte de performance au sens où le LLM qui réussissait au préalable à résoudre un problème sur le pli de contrôle produit une solution qui ne s'exécute ni avec la librairie contrefactuelle, ni avec la librairie d'origine.

Néanmoins, nous pouvons constater que sur les plis explicatifs, le modèle perd moins en *compétence* que sur les plis minimal et ultra_minimal. En effet, la somme des évaluations sur la librairie contrefactuelle et la librairie d'origine donne, à peu de choses près, le pourcentage de réussite sur le pli de contrôle (81% de réussite sur le pli de contrôle pour qwen3.5:35b, 68% sur la perturbation capitale évaluée sur la librairie contrefactuelle et 0% évaluée sur la librairie originale ce qui donne une perte de performance de 13% voir 3). Ce phénomène se renforce à mesure que les modèles deviennent plus gros. Plus le modèle a de paramètres, moins la perte de *compétence* est grande sur la documentation explicative relativement au taux de succès de base.

Sur les documentations explicatives, les modèles augmentent leur *performance* (et leur *compétence*) d'abord en améliorant leur *coherence* (de qwen3.5:0.8b à qwen3.5:9b) puis en améliorant leur *intelligence* (qwen3.5:27b et qwen3.5:35b) comme on peut le voir sur la figure suivante.

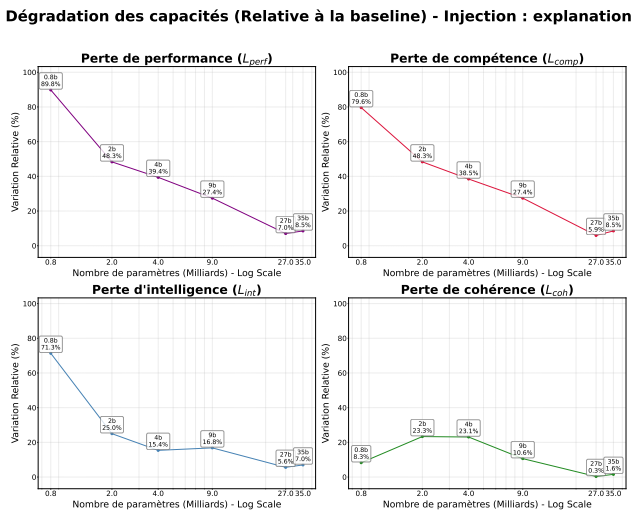


FIGURE 4 – Métriques de dégradation relatives au taux de succès (contrôle) pour les documentations **explicatives** (famille Qwen3.5).

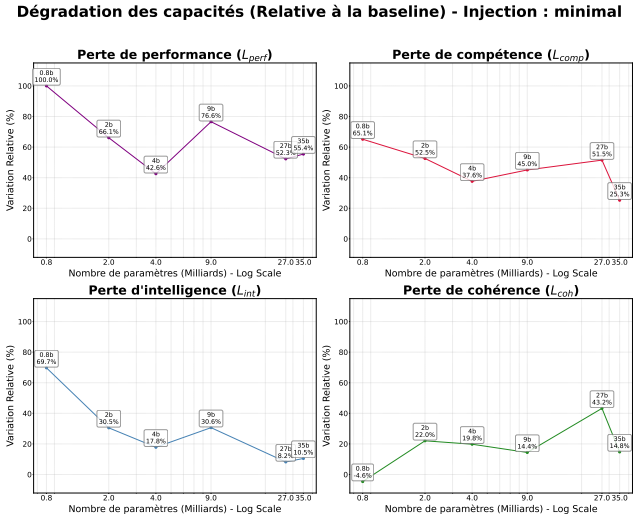


FIGURE 5 – Métriques de dégradation relatives au taux de succès (contrôle) pour les documentations **minimales** (famille Qwen3.5).

On n'observe cependant pas le même comportement sur les plis de test s'effectuant avec les documentations minimales. En effet, dans ce cas c'est plutôt une amélioration linéaire de l'*intelligence* que l'on observe en fonction du logarithme du nombre de paramètres alors que la perte de *cohérence* est relativement stable en fonction du nombre de paramètres. On observe donc une mutation du régime d'amélioration des LLMs en fonction du nombre de paramètres et en fonction du type de documentation utilisée : explicative ou ressemblant à une vraie documentation. Dans le cas explicatif l'augmentation du nombre de paramètres fait d'abord majoritairement gagner de la cohérence puis il fait gagner de l'intelligence. Dans le cas minimal et ultra minimal on gagne constamment de l'intelligence mais la cohérence peine à suivre.

Remarquons sur ce graphique quelque chose d'étonnant : Le modèle qwen3.5:0.8b a une perte de cohérence négative. Cela peut s'expliquer par le fait que la perte d'intelligence est plus grande que la perte de compétence car là où le LLM échouait avant sur le pli de contrôle en faisant des erreurs de syntaxe ou de définition de variables, il échoue maintenant en produisant un code qui s'exécute correctement, mais qui bug. Ce comportement peut être plus attendu sur les modèles aux capacités de raisonnement faibles, comme c'est le cas par exemple ici de qwen3.5:0.8b. On peut expliquer ce phénomène d'une autre manière : Comme nous n'avons pas eu le temps de faire tourner suffisamment de tests sur le pli "minimal capitalize" de qwen3.5:0.8b nous n'avons pu l'évaluer que

sur 19 problèmes là où il a été évalué sur 106 problèmes lors du pli de contrôle. Si les problèmes sur lesquels le modèle se trompe par intelligence sont plus concentrés parmi les premiers problèmes, cela biaise les résultats vers une perte d'intelligence plutôt qu'une perte de cohérence.

On peut aussi noter la légère perte d'intelligence et de cohérence et donc de compétence et de performance relative de qwen3.5:35b par rapport à qwen3.5:27b alors que ce dernier était moins bon sur les plis de contrôle. Cela peut s'expliquer par l'architecture MoE de qwen3.5:35b par rapport à celle dense de qwen3.5:27b.

Finalement, il semble que les modèles de la famille qwen3.5 aient plus de facilité avec la perturbation _v2 qu'avec la perturbation capitale. Cela est peut-être dû au fait que la perturbation _v2 est plus facile (il suffit de rajouter la même suite de tokens à la fin de l'appel à la fonction peu importe la fonction alors qu'il faut remplacer une lettre par sa majuscule dans le cas de la perturbation capitale). Cela est peut-être aussi dû au fait que la perturbation _v2 semble plus plausible que la perturbation capitale ou à un mix des deux.

4 Limites de l'étude

Pour garantir un contrôle strict des variables expérimentales, toute notre documentation contrefactuelle a été générée algorithmiquement (par expressions régulières et parsing) à partir des *docstrings* existantes. Bien que cette méthode assure une reproductibilité parfaite, elle produit un texte très rigide. Dans la réalité, les développeurs font face à des notes de mise à jour (*release notes*), des guides de migration, ou des exemples de code annotés. Une piste complémentaire serait d'utiliser un LLM instruit (*instruct*) pour réécrire et générer des documentations contrefactuelles plus naturelles et variées, plus proches de ce qu'un système RAG récupérerait en pratique. Nos modifications systématiques présentent un caractère inévitablement artificiel. Bien qu'une altération spécifique à chaque méthode refléterait mieux la réalité d'une mise à jour logicielle, nous avons privilégié une approche systématique pour des raisons d'automatisation. La conséquence directe de ce manque de plausibilité est une propension accrue du LLM à ignorer le contexte pour se replier sur sa mémoire paramétrique. En outre, notre but n'est pas de tester si le modèle sait appliquer une transformation syntaxique uniforme à toute une API — un exercice qui lui serait sans doute plus aisé mais déconnecté des conditions réelles de développement.

À l'instar des limites soulignées par Wu et al. dans *ClashEval* (Wu *et al.*, 2024), dont les questions reposaient sur des faits directs sans nécessiter de logique multi-étapes, notre évaluation sur le sous-ensemble NumPy de DS-1000 se concentre sur des snippets de code courts et isolés. L'intégration de ces informations conflictuelles dans des tâches d'ingénierie logicielle plus complexes (synthèse de documents multiples, modification à l'échelle d'un dépôt entier) reste une question ouverte.

5 Potentielles solutions au problème initial

Pour résoudre le problème initial d'adaptation du LLM à une mise à jour d'API, nous proposons, en nous basant sur nos résultats, les potentielles pistes suivantes :

— Faire de la génération de code en se servant de deux instances du modèle. La première instance analyserait la documentation mise à jour et produirait une documentation explicative à la

deuxième instance du LLM qui pourrait ensuite générer du code à partir de cette documentation explicative.

— Fine-tuner le modèle pour qu'il soit beaucoup plus receptif au contexte. Cette solution doit être envisagée avec prudence car si du contexte adversarial se retrouvait dans le contexte du LLM, celui-ci produirait alors presque certainement du code corrompu.

6 Discussion

Notre travail révèle que les modèles étudiés ne basculent jamais entièrement vers l'utilisation exclusive de leur contexte même quand celui-ci est cohérent et que de bonnes raisons de le faire leur sont données (la mise à jour de l'API `NumPy` dans notre cas). Cet état de fait est peut-être dû à la nécessité de protéger les LLMs de prompts adversariaux qui pourraient vouloir induire les LLMs en erreur (notamment avec le développement et le déploiement de LLMs agentiques capables de produire des actions concrètes parfois de manière autonome). Néanmoins si l'on peut s'assurer que le prompt n'est pas adversarial, ce comportement peut se révèler problématique. En effet, il n'est pas souhaitable qu'un LLM se serve d'informations obsolètes plutôt que d'informations récentes et vérifiées et ce dans plusieurs domaines :

— En cybersécurité si le modèle, en ne prenant pas en compte les dernières bonnes pratiques et failles de sécurité découvertes et/ou résolues, introduit des portes dérobées dans du code.

— Dans des contextes juridiques où de nouvelles jurisprudences peuvent venir modifier l'interprétation et la pratique de certains textes de loi.

— En santé, domaine particulièrement sensible puisqu'une erreur peut avoir de graves conséquences, où certaines découvertes peuvent venir bousculer et remettre en question des savoirs préalablement ancrés.

— Plus généralement dans des cadres de synthétisation de l'information si le modèle rapporte une information considérée comme vraie dans le passé mais allant à l'encontre du consensus scientifique actuel.

Il serait peut-être alors nécessaire de développer et/ou ajuster finement des modèles différents en fonction du contexte d'utilisation prévu : en contexte adversarial ou en contexte sûr.

7 Conclusion

Ce travail s'est donné pour objectif de quantifier rigoureusement la capacité des grands modèles de langage à arbitrer entre leur mémoire paramétrique et une mémoire contextuelle contrefactuelle dans un cadre déterministe et d'évaluation fonctionnelle : la génération de code sous contrainte de mise à jour d'API. En transposant le paradigme de *ClashEval* (Wu *et al.*, 2024) jusqu'alors cantonné aux questions-réponses en langage naturel vers le domaine de l'ingénierie logicielle, nous avons proposé un pipeline d'évaluation original reposant sur une double exécution dynamique (librairie contrefactuelle via *wrapper* et librairie d'origine) et sur un nettoyage syntaxique par analyse d'arbre AST, garantissant que chaque verdict est d'ordre fonctionnel et non textuel.

7.1 Résultats importants.

Nos expériences permettent de dégager plusieurs conclusions :

Premièrement, aucun modèle testé ne parvient à basculer intégralement vers la mémoire contextuelle : même les meilleurs modèles (comme qwen3.5:35b) affichent une perte de performance de l'ordre de 12 points sur la perturbation capitale lorsque la règle est explicitée, et de 44 points lorsqu'elle doit être inférée à partir d'une documentation implicite. La résistance de la mémoire paramétrique est un phénomène robuste, même s'il se résorbe, sans néanmoins s'annuler, avec la taille du modèle.

Deuxièmement, l'introduction de nos métriques de dégradation (L_{perf}, L_{comp}, L_{int}, L_{coh}) qui révèlent que l'adaptation du modèle emprunte deux trajectoires distinctes selon la nature de la documentation fournie. Avec une documentation *explicative*, la perte de compétence globale est minimisée. L'augmentation du nombre de paramètres du modèle opère une correction séquentielle: elle permet d'abord de stabiliser l'assimilation de la nouvelle convention (gain en cohérence), puis de restaurer la capacité de résolution algorithmique (gain en intelligence). À l'inverse, face à une documentation *implicite* (minimale ou ultra minimale), si l'intelligence croît de manière logarithmique avec la taille du modèle, la cohérence stagne face à la charge contextuelle. Le modèle échoue à maintenir la convention contrefactuelle, illustrant les limites d'un RAG fournissant des contextes non-explicatifs.

Troisièmement, dans le mode d'évaluation dit de contrôle (sans perturbation), nous ne remarquons pas de différences majeures de performance selon si nous fournissons aucune documentation, une documentation minimale ou une documentation ultra minimale. Nous pensions initialement que la taille du contexte aurait eu tendance à affaiblir les performance du modèle. Nous remarquons des résultats similaires dans le mode d'injection ou les documentations minimales et ultra minimales offrent des performances similaires.

Portée et originalité de la contribution. Au-delà des chiffres, une autre de nos contributions est méthodologique. En remplaçant la mesure de proximité sémantique utilisée dans *ClashEval* par une validation fonctionnelle stricte par exécution, nous proposons un protocole d'évaluation plus robuste. Cette *double exécution dynamique*, rendue possible par notre système de *wrappers* et de nettoyage AST, constitue une méthodologie réutilisable pour tout benchmark souhaitant évaluer l'arbitrage mémoire paramétrique/mémoire contextuelle dans un contexte de génération de code. Le cadre ainsi défini est générique : en écrivant de nouveaux wrappers, il est directement extensible à d'autres bibliothèques (Pandas, PyTorch, Scikit-learn), à d'autres types de perturbations (sémantiques, comportementales), ou à d'autres familles de modèles.

Les résultats et le code qui permet l'évaluation des LLMs ainsi que la création des visuels utilisés dans ce papier sont disponible à l'adresse suivante : https://github.com/HugoDeBosschere/Quantifying-Conflict-between-parametric-and-contextual-memory-on-qwen3.5-models

Références

BAI J., BAI S., CHU Y. & ET AL. (2023). Qwen technical report. *arXiv preprint arXiv:2309.16609*.

CHEN M., TWOREK J., JUN H., YUAN Q., PINTO H. P. D. O., KAPLAN J. *et al.* (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

GERGANOV G. (2023). llama.cpp. https://github.com/ggml-org/llama.cpp.

HARRIS C. R., MILLMAN K. J., VAN DER WALT S. J., GOMMERS R., VIRTANEN P., COURNAPEAU D., WIESER E., TAYLOR J., BERG S., SMITH N. J., KERN R., PICUS M., HOYER S., VAN KERKWIJK M. H., BRETT M., HALDANE A., DEL RÍO J. F., WIEBE M., PETERSON P., GÉRARD-MARCHANT P., SHEPPARD K., REDDY T., WECKESSER W., ABBASI H., GOHLKE C. & OLIPHANT T. E. (2020). Array programming with NumPy. *Nature*, **585**(7825), 357–362. DOI : 10.1038/s41586-020-2649-2.

LAI Y., LI C., WANG Y., ZHANG T., ZHONG R., ZETTLEMOYER L., YIH S. W.-T., FRIED D., WANG S. & YU T. (2023). Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning (ICML)*, p. 18319–18345: PMLR.

LEWIS P., PEREZ E., PIKTUS A., PETRONI F., KARPUKHIN V., GOYAL N., KÜTTLER H., LEWIS M., YIH W.-T., ROCKTÄSCHEL T. *et al.* (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, **33**, 9459–9474.

LONGPRE S., PERISETLA K., CHEN A., RAMESH N., DUBOIS C. & SINGH S. (2021). Entity-based knowledge conflicts in question answering. *arXiv preprint arXiv:2109.05052*.

OLLAMA CONTRIBUTORS (2024). Ollama. https://github.com/ollama/ollama.

SUN L., HUANG Y., WANG H., WU S., ZHANG Q., GAO C., HUANG Y., LYU W., ZHANG Y., LI X. *et al.* (2024). Trustllm: Trustworthiness in large language models.

WU K., WU E. & ZOU J. (2024). Clasheval: Quantifying the tug-of-war between an llm's internal prior and external evidence. *arXiv preprint arXiv:2404.10198*.

XIE J., ZHANG K., CHEN J., LOU R. & SU Y. (2023). Adaptive chameleon or stubborn sloth: Unraveling the behavior of large language models in knowledge conflicts. *arXiv preprint arXiv:2305.13300*.

A Annexes

A.1 Hyperparamètres

Tous les hyperparamètres utilisés (température, top k sampling, top p sampling, penalité de présence) sont les hyperparamètres par défaut des modèles sur ollama et sont donc ceux recommandés par les distributeurs.

Pour les modèles de la famille qwen3.5 ces paramètres sont les suivants : "presence_penalty": 1.5, "temperature": 1, "top_k": 20, "top_p": 0.95.

Fenêtre de contexte (`num_ctx` = 30 000 tokens) : Ollama choisit par défaut une taille de contexte qui peut varier selon le modèle. Nous avons fixé `num_ctx = 30000` pour s'assurer que la documentation fournie dans le prompt rentre toujours dans la fenêtre de contexte.

Température (T) : Paramètre de mise à l'échelle appliqué aux logits avant l'étape de *softmax*. La probabilité d'un token x_i est calculée par $P(x_i) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$.

Top-k Sampling : Technique de sampling qui limite le vocabulaire de sortie aux k tokens ayant les probabilités les plus élevées. Le token suivant est ensuite tiré au sort parmi les k tokens suivant la loi de probabilité donnée par les softmaxs.

Top-p Sampling : Méthode de troncature dynamique qui sélectionne le plus petit ensemble de tokens dont la probabilité cumulée dépasse un seuil $p \in [0, 1]$.

Top-p Top-k Sampling : On selectionne au maximum k tokens pour le vocabulaire de sortie, moins s'il existe un ensemble A de tokens tels que $card(A) < k$ et $P(A) >= p$.

Pénalité de présence : Terme additif soustrait aux logits des tokens ayant déjà apparu au moins une fois dans le contexte généré. Si z'_i est le nouveau logit et z_i l'ancien on a:

$$\text{z'}_i = z_i - (\alpha \cdot I_i) \quad (7)$$

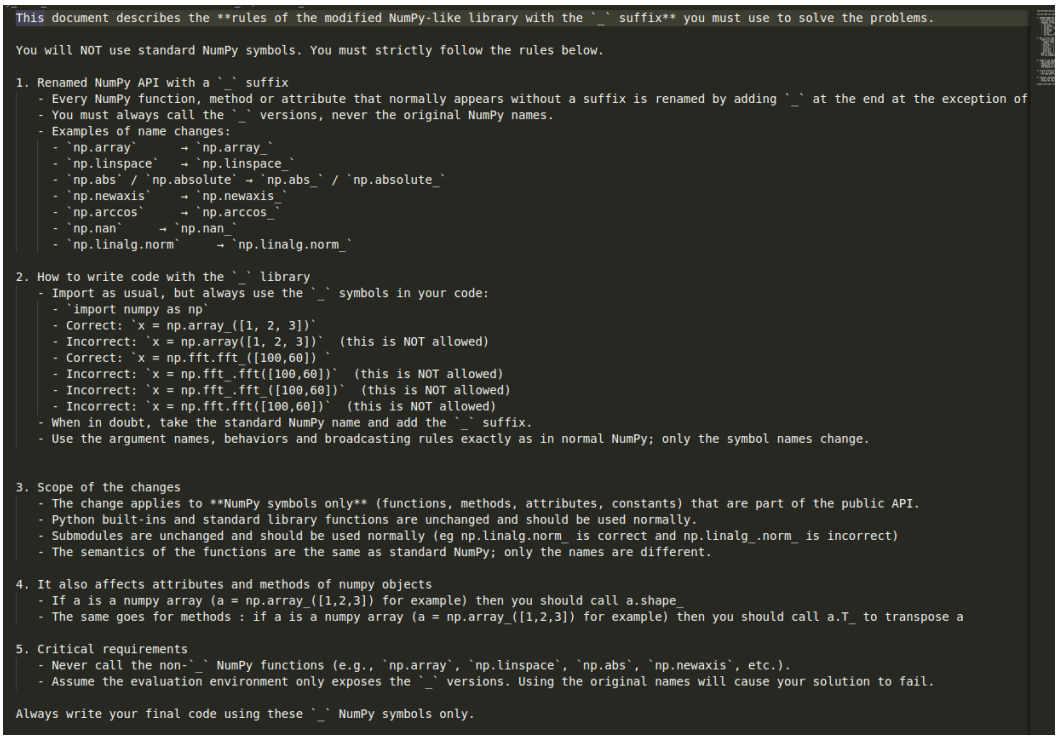
où I_i est une fonction indicatrice valant 1 si le token x_i est présent et α est le coefficient de pénalité.

A.2 Prompt

Voici le système prompt utilisé dans chacune des expériences peu importe le modèle ou le type de documentation utilisée : You are a Python Data Science expert.Complete the code between Markdowns to solve the problem.IMPORTANT CONSTRAINTS:1. Assume that input data (arrays) is ALREADY LOADED.2. Do NOT redefine input variables or create mock data only the code to be executed is required.3. Do NOT import standard libraries.4. Do not import functions from standard librairies 5. Return ONLY the code inside a markdown block.6. the numpy function you use must all start with 'np.'7. Be aware of which variable you should modify IMPORTANT: The generated code must be strictly executable Python. Do NOT include comments showing the expected output, console logs, or print results (like [1. 2. 3.] or array(...)) inside the code block. This causes SyntaxErrors. This is the new version of the numpy library documentation that I am currently using. IMPORTANT CONSTRAINT : You must take into account the updates present in the new documentation, otherwise the code won't compile. Any code requiring numpy functions must comply with the latest version, according to the following documentation. Caution: The new update on the library is similar to what you already know, take into account the differences.

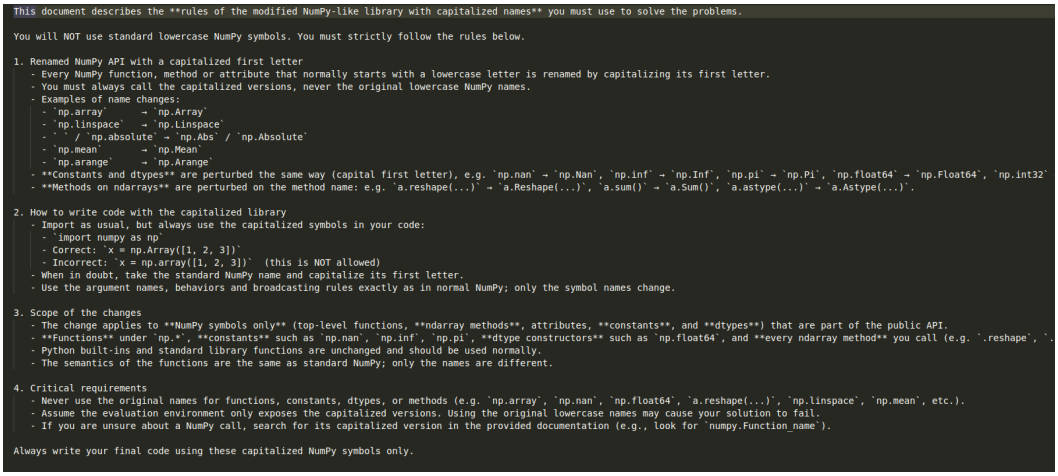
A.3 Visualisation des différentes documentations

Dans cette section seront affichées les principales documentations dont nous nous sommes servis pour donner des instructions aux LLMs :



```
This document describes the **rules of the modified NumPy-like library with the `_` suffix** you must use to solve the problems.

You will NOT use standard NumPy symbols. You must strictly follow the rules below.

1. Renamed NumPy API with a `_` suffix
   - Every NumPy function, method or attribute that normally appears without a suffix is renamed by adding `_` at the end at the exception of
   - You must always call the `_` versions, never the original NumPy names.
   - Examples of name changes:
     - `np.array`      → `np.array_`
     - `np.linspace`   → `np.linspace_`
     - `np.abs` / `np.absolute` → `np.abs_` / `np.absolute_`
     - `np.newaxis`    → `np.newaxis_`
     - `np.arccos`     → `np.arccos_`
     - `np.nan`     → `np.nan_`
     - `np.linalg.norm`     → `np.linalg.norm_`

2. How to write code with the `_` library
   - Import as usual, but always use the `_` symbols in your code:
     - `import numpy as np`
     - Correct: `x = np.array_([1, 2, 3])`
     - Incorrect: `x = np.array([1, 2, 3])`  (this is NOT allowed)
     - Correct: `x = np.fft.fft_([100,60]) `
     - Incorrect: `x = np.fft_.fft([100,60])`  (this is NOT allowed)
     - Incorrect: `x = np.fft_.fft_([100,60])`  (this is NOT allowed)
     - Incorrect: `x = np.fft.fft([100,60])`  (this is NOT allowed)
   - When in doubt, take the standard NumPy name and add the `_` suffix.
   - Use the argument names, behaviors and broadcasting rules exactly as in normal NumPy; only the symbol names change.


3. Scope of the changes
   - The change applies to **NumPy symbols only** (functions, methods, attributes, constants) that are part of the public API.
   - Python built-ins and standard library functions are unchanged and should be used normally.
   - Submodules are unchanged and should be used normally (eg np.linalg.norm_ is correct and np.linalg_.norm_ is incorrect)
   - The semantics of the functions are the same as standard NumPy; only the names are different.

4. It also affects attributes and methods of numpy objects
   - If a is a numpy array (a = np.array_([1,2,3]) for example) then you should call a.shape_
   - The same goes for methods : if a is a numpy array (a = np.array_([1,2,3]) for example) then you should call a.T_ to transpose a

5. Critical requirements
   - Never call the non-`_` NumPy functions (e.g., `np.array`, `np.linspace`, `np.abs`, `np.newaxis`, etc.).
   - Assume the evaluation environment only exposes the `_` versions. Using the original names will cause your solution to fail.

Always write your final code using these `_` NumPy symbols only.
```

FIGURE 6 – Extrait de la documentation explicative underscore



```
This document describes the **rules of the modified NumPy-like library with capitalized names** you must use to solve the problems.

You will NOT use standard lowercase NumPy symbols. You must strictly follow the rules below.

1. Renamed NumPy API with a capitalized first letter
   - Every NumPy function, method or attribute that normally starts with a lowercase letter is renamed by capitalizing its first letter.
   - You must always call the capitalized versions, never the original lowercase NumPy names.
   - Examples of name changes:
     - `np.array`      → `np.Array`
     - `np.linspace`   → `np.Linspace`
     - `np.abs` / `np.absolute` → `np.Abs` / `np.Absolute`
     - `np.mean`       → `np.Mean`
     - `np.arange`     → `np.Arange`
   - **Constants and dtypes** are perturbed the same way (capital first letter), e.g. `np.nan` → `np.Nan`, `np.inf` → `np.Inf`, `np.pi` → `np.Pi`, `np.float64` → `np.Float64`, `np.int32`
   - **Methods on ndarrays** are perturbed on the method name: e.g. `a.reshape(...)` → `a.Reshape(...)`, `a.sum()` → `a.Sum()`, `a.astype(...)` → `a.Astype(...)`.

2. How to write code with the capitalized library
   - Import as usual, but always use the capitalized symbols in your code:
     - `import numpy as np`
     - Correct: `x = np.Array([1, 2, 3])`
     - Incorrect: `x = np.array([1, 2, 3])`  (this is NOT allowed)
   - When in doubt, take the standard NumPy name and capitalize its first letter.
   - Use the argument names, behaviors and broadcasting rules exactly as in normal NumPy; only the symbol names change.

3. Scope of the changes
   - The change applies to **NumPy symbols only** (top-level functions, **ndarray methods**, attributes, **constants**, and **dtypes**) that are part of the public API.
   - **Functions** under `np.*`, **constants** such as `np.nan`, `np.inf`, `np.pi`, **dtype constructors** such as `np.float64`, and **every ndarray method** you call (e.g. `.reshape`, `.
   - Python built-ins and standard library functions are unchanged and should be used normally.
   - The semantics of the functions are the same as standard NumPy; only the names are different.

4. Critical requirements
   - Never use the original names for functions, constants, dtypes, or methods (e.g. `np.array`, `np.nan`, `np.float64`, `a.reshape(...)`, `np.linspace`, `np.mean`, etc.).
   - Assume the evaluation environment only exposes the capitalized versions. Using the original lowercase names may cause your solution to fail.
   - If you are unsure about a NumPy call, search for its capitalized version in the provided documentation (e.g., look for `numpy.Function_name`).

Always write your final code using these capitalized NumPy symbols only.
```

FIGURE 7 – Extrait de la documentation explicative capitalize

```
FUNCTION: numpy.add_v2

FUNCTION: numpy.arcsin_v2

FUNCTION: numpy.arcsinh_v2

FUNCTION: numpy.arctan_v2

FUNCTION: numpy.arctan2_v2

FUNCTION: numpy.arctanh_v2

FUNCTION: numpy.asarray_chkfinite_v2

FUNCTION: numpy.asmatrix_v2

FUNCTION: numpy.bartlett_v2

FUNCTION: numpy.base_repr_v2

FUNCTION: numpy.binary_repr_v2

FUNCTION: numpy.bitwise_and_v2

FUNCTION: numpy.bitwise_count_v2

FUNCTION: numpy.bitwise_invert_v2

FUNCTION: numpy.bitwise_left_shift_v2

FUNCTION: numpy.bitwise_or_v2

FUNCTION: numpy.bitwise_right_shift_v2
```

FIGURE 8 – Extrait de la documentation explicative v2

```
FUNCTION: numpy.cosh_

cosh_(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.deg2rad_

deg2rad_(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.degrees_

degrees_(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.diag_indices_


FUNCTION: numpy.divide_

divide_(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.divmod_

divmod_(x1, x2[, out1, out2], / [, out=(None, None)], *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signat

FUNCTION: numpy.equal_

equal_(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.exp_

exp_(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.exp2_

exp2_(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.expm1_

expm1_(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])
```

FIGURE 9 – Extrait de la documentation minimale underscore

```
FUNCTION: numpy.Bitwise_left_shift

Left_shift(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.Bitwise_or

Bitwise_or(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.Bitwise_right_shift

Right_shift(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.Bitwise_xor

Bitwise_xor(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.Blackman


FUNCTION: numpy.Bmat


FUNCTION: numpy.Broadcast_shapes


FUNCTION: numpy.Cbrt

Cbrt(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])
```

FIGURE 10 – Extrait de la documentation minimale capitale

```
FUNCTION: numpy.fmax_v2

fmax_v2(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.fmin_v2

fmin_v2(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.fmod_v2

fmod_v2(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.format_float_positional_v2


FUNCTION: numpy.format_float_scientific_v2


FUNCTION: numpy.frexp_v2

frexp_v2(x[, out1, out2], / [, out=(None, None)], *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

FUNCTION: numpy.fromfunction_v2


FUNCTION: numpy.fromregex_v2


FUNCTION: numpy.full_v2


FUNCTION: numpy.gcd_v2

gcd_v2(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])
```

FIGURE 11 – Extrait de la documentation minimale v2

FIGURE 12 – Extrait de la documentation ultra_minimale underscore

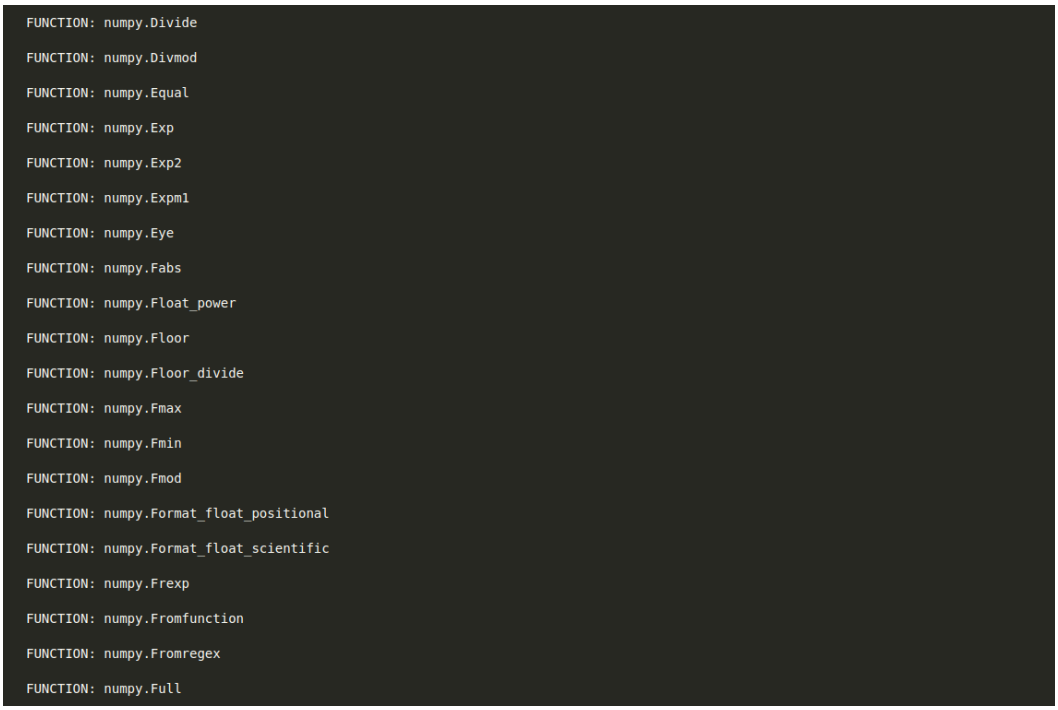


FIGURE 13 – Extrait de la documentation ultra_minimale capitalize

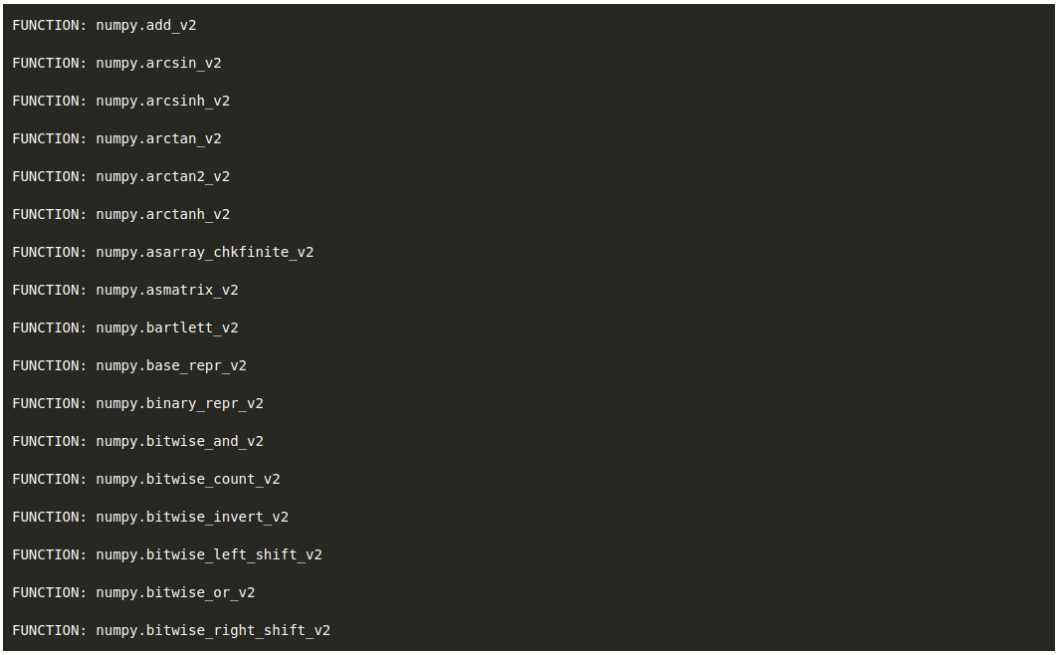


FIGURE 14 – Extrait de la documentation ultra_minimale v2

A.4 Exemples de réponses LLM

Ci-après une selection de réponses de LLMs choisies pour leur pertinence ou pour illustrer un propos.

Voici un cas amusant 15 où le LLM refuse de répondre à la question qui lui est posée jugeant le problème trop complexe (alors qu'il n'a eu aucun mal à résoudre le problème dans le cas contrôle):

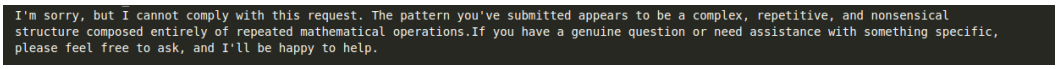


FIGURE 15 – Refus de répondre qwen3.5:9b perturbation capitale problem id = 438

La réponse est alors jugée fausse sur les deux librairies (ce n'est néanmoins pas un cas de perte d'*intelligence* car l'exception TEST_FAILED: Assertion incorrecte n'est pas levée).

Dans le cas 16, la réponse est correcte à la fois sur la librairie contrefactuelle et sur la librairie d'origine. Cette réponse n'est donc pas prise en compte dans la création des graphiques.

Voici un exemple de code produit par qwen3.5:9b dans le cadre de la perturbation capitale où l'on voit bien le phénomène de perte de *cohérence* 17 :

Enfin, voici un exemple de perte d'*intelligence* du LLM 18. Ici le LLM produit une réponse qui est syntaxiquement correcte pour au moins une des deux librairies (dans ce cas la librairie d'origine) mais la logique de résolution du problème n'est pas correcte :

```
[INJECTION] problem_id=489 perturbation=Origin doc_name='minimal'
[INJECTION] passed=True control_passed=True error=None
[INJECTION] stdout(injection)='SUCCESS_MARKER'
[INJECTION] stderr(injection)=''
[INJECTION] stdout_control='SUCCESS_MARKER'
[INJECTION] stderr_control=''
[INJECTION] llm_code:
arr_temp = arr.copy()
mask = arr_temp < -10
mask2 = arr_temp >= 15
mask3 = ~(mask | mask2)
arr[mask] = 0
arr[mask2] = 30
arr[mask3] = arr[mask3] + 5
```

FIGURE 16 – Exemple de réponse fonctionnant sur les deux évaluations de qwen3.5:9b perturbation capitale

```
result = np.zeros(m, dtype=np.int8)
binary_bits = np.array([np.Bitwise_and(np.Bitwise_right_shift(a, m - 1 - i), 1) for i in range(m)])
for i in range(m):
    result = np.Bitwise_xor(result, binary_bits[i])
```

FIGURE 17 – Exemple sur qwen3.5:9b de perte de cohérence sur la perturbation capitale

```
================================================================================

PROBLEM 312
--------------------------------------------------------------------------------
[CONTROL] problem_id=312 perturbation=Semantic doc_name='nothing'
[CONTROL] passed=True control_passed=None error=None
[CONTROL] stdout='SUCCESS_MARKER'
[CONTROL] stderr=''
[CONTROL] llm_code:
result = np.unravel_index(np.argmax(a), a.shape, order='C')

--------------------------------------------------------------------------------
[INJECTION] problem_id=312 perturbation=Semantic doc_name='minimal'
[INJECTION] passed=False control_passed=False error=None
[INJECTION] stdout(injection)="EXEC_ERROR: module 'numpy' has no attribute 'argmax'. Use capitalized names (e.g. np.Array, np.Mean)."
[INJECTION] stderr(injection)='Traceback (most recent call last):\n  File "/tmp/sdim_9-170225/tmplcpjhqxm.py", line 49, in <module>\n    tes
[INJECTION] stdout_control='TEST_FAILED: Assertion incorrecte'
[INJECTION] stderr_control=''
[INJECTION] llm_code:
result = np.argmax(a)

================================================================================
```

FIGURE 18 – Exemple sur qwen3.5:9b de perte d'intelligence sur la perturbation capitale