# L’art du **prompt engineering :** des bases aux techniques avancées

Jean-Claude LAROCHE & Fabrice REBY

GROUPE
DIGITAL MAKERS

# L’art du **prompt engineering :** des bases aux techniques avancées

Jean-Claude LAROCHE & Fabrice REBY

# Préface

Lorsque l'on a consacré sa carrière à naviguer au cœur des transformations numériques des grandes entreprises, on développe une acuité particulière pour identifier les véritables points de rupture technologique. L'émergence des modèles de langage de grande taille (LLM), et au-delà, l'intelligence artificielle dite « agentique », constituent indéniablement l'une de ces inflexions majeures, ouvrant un champ d'opportunités et de questionnements d'une ampleur inédite.

En tant qu'ancien président d'une association regroupant les plus importantes organisations françaises utilisatrices de solutions numériques (le Cigref), et fort de mon expérience en tant que DSI de grandes entreprises (le Groupe EDF, puis Enedis), j'ai eu le privilège d'être aux premières loges de plusieurs révolutions technologiques. Rares sont celles qui ont suscité autant de potentiel transformateur.

C'est pourquoi j'ai eu le plaisir de coécrire cet ouvrage avec Fabrice Reby (Président fondateur d'Oxyl), un professionnel dont j'apprécie depuis longtemps la vision, la précision analytique et la capacité à structurer les sujets complexes de manière accessible. Nos parcours, bien que différents, se sont souvent croisés autour d'un objectif commun : rendre les technologies compréhensibles et utiles pour les organisations. Ensemble, nous avons voulu transmettre non seulement des connaissances, mais une démarche intellectuelle, ancrée dans l'expérience du terrain.

Fabrice et moi partageons une même exigence : celle de l'intelligibilité et de l'utilité opérationnelle, en intégrant le contexte de menaces climatiques, de tensions géopolitiques, d'inquiétudes sur le devenir humain. Nous savons que les professionnels, qu'ils soient décideurs, ingénieurs, consultants ou enseignants, attendent des repères clairs et des méthodes concrètes. Ce livre s'adresse à eux, mais aussi à toute personne curieuse de comprendre comment converser efficacement avec les grands modèles de langage, au-delà de l'effet de mode.

Ce projet s'inscrit dans une époque où les compétences humaines et l'intelligence artificielle doivent apprendre à co-construire de nouveaux équilibres. Le prompt engineering ne se réduit pas à « bien parler à une machine » ; il s'agit d'une discipline transversale, mêlant langage, logique et stratégie. C'est tout l'enjeu de ce travail : offrir une grille de lecture claire et évolutive de cette pratique émergente.

Ce livre est donc né d'une conviction profonde : le prompt engineering représente le premier pas, essentiel et structurant, sur le vaste chemin de l'intégration de l'intelligence artificielle au sein de nos organisations. Nous avons ainsi souhaité rédiger ce document avec une approche résolument pédagogique, afin de vous transmettre les bases indispensables à une compréhension solide. Car loin d'être une simple tendance, le prompt engineering s'impose comme une compétence clé de la nouvelle ère numérique. C'est la grammaire qui nous permettra non seulement de converser avec l'intelligence artificielle, mais de la guider, de l'orienter et d'en libérer tout son potentiel au service de nos organisations.

Je remercie chaleureusement Fabrice pour la qualité et la richesse des échanges qui nourrissent notre collaboration. Ceux-ci ont donné l'idée de co-signer cet ouvrage, montrant ainsi la convergence des préoccupations entre décideurs et experts, lorsqu'émerge une technologie aussi transformatrice que l'intelligence artificielle.

Je vous invite à vous plonger dans ces pages avec curiosité et l'esprit ouvert. Vous y trouverez non seulement des connaissances précieuses, mais également une démarche intellectuelle rigoureuse pour aborder l'avenir avec lucidité et confiance.
Bonne lecture.

**Jean-Claude LAROCHE**

# Sommaire

# Introduction

Les avancées récentes en intelligence artificielle, notamment dans le domaine des modèles de langage génératifs (LLM – Large Language Models), ont radicalement transformé la manière dont nous interagissons avec la technologie. Ces modèles permettent des échanges en langage naturel, remplaçant ainsi les instructions rigides par des conversations fluides et intuitives. Grâce à leur capacité à traiter des instructions variées et complexes, ces outils ouvrent des perspectives sans précédent dans de nombreux domaines, de la recherche à l'assistance personnelle, en passant par la création de contenu automatisée.

Cependant, pour tirer pleinement parti de ces puissants modèles d'IA, une approche spécialisée est indispensable : le **prompt engineering**. Cette technique consiste à formuler des requêtes ou des instructions de manière à maximiser la pertinence et la qualité des réponses obtenues. L'art de rédiger un prompt clair et précis est essentiel pour obtenir des résultats de haute qualité.

Ce livre vous guidera à travers les meilleures pratiques et techniques avancées du prompt engineering, que vous soyez déjà utilisateur ou simplement curieux des dernières avancées en IA. Cependant, la meilleure manière de maîtriser le prompt engineering est de pratiquer au fur et à mesure de cette lecture. Utilisez différents LLM, expérimentez avec des prompts variés, et n'oubliez pas que la pratique est la clé pour développer une véritable expertise. En vous exerçant régulièrement, vous découvrirez comment affiner vos requêtes pour obtenir des résultats de plus en plus précis et adaptés à vos besoins.

# /Chapitre 1
# Comprendre les Grands Modèles de Langage – Un voyage au cœur des LLM

Avant de plonger dans les techniques du *prompt engineering*, il est essentiel de comprendre comment fonctionnent réellement ces modèles de langage. Comprendre leurs mécanismes internes, leurs forces et leurs limitations, est la clé pour éviter de se contenter d'appliquer des « recettes » de *prompts* de manière aléatoire, et au contraire développer une véritable intuition et une maîtrise de cet art. Ce chapitre est donc une invitation à un voyage au cœur des *LLM*, pour en décortiquer les rouages et poser des bases solides à notre exploration du *prompt engineering*.

## 1.1 Les tokens : les briques élémentaires du langage pour l'IA

Lorsque nous communiquons avec un *LLM*, que ce soit pour lui donner un *prompt* ou pour qu'il nous réponde, le texte n'est pas traité tel quel. En réalité, les *LLM* travaillent avec une représentation plus fondamentale du langage : les *tokens*.

Imaginez les *tokens* comme les briques **LEGO** du langage. Ce sont les unités de base avec lesquelles les *LLM* construisent leur compréhension et leur génération de texte. Mais attention, un *token* n'est pas toujours équivalent à un mot entier. Un *token* peut être :

- Un mot entier : « chat », « ordinateur », « intelligence »
- Une partie de mot : « dé- » (dans « démarrer »), « -ment » (dans « rapidement »), « ing » (dans « running »)
- Un caractère spécial : , , ., !, l'espace ou d'autres symboles de ponctuation

Pourquoi cette découpe en *tokens* ? Pourquoi ne pas simplement travailler avec des mots entiers ? L'utilisation de *tokens* apporte plusieurs avantages cruciaux pour les *LLM* :

- **Vocabulaire restreint et efficace :** Au lieu de devoir mémoriser et traiter des centaines de milliers, voire des millions de mots différents, le *LLM* travaille avec un vocabulaire limité de *tokens* (souvent de l'ordre de quelques dizaines de milliers). C'est comme apprendre à construire des structures complexes avec un nombre limité de briques LEGO : on gagne en efficacité.
- **Meilleure gestion des mots rares ou inconnus :** Si un mot rare ou inconnu du modèle apparaît dans un texte, il peut être décomposé en une séquence de *tokens* plus fréquents que le modèle connaît. Cela permet de traiter un vocabulaire potentiellement illimité, même avec un vocabulaire de *tokens* restreint.
- **Flexibilité multilingue :** La *tokenisation* peut être adaptée à différentes langues et systèmes d'écriture. Un même modèle peut ainsi traiter plusieurs langues en utilisant des *tokenizers* spécifiques à chaque langue.

Concrètement, comment un texte est-il transformé en *tokens* ? Un processus appelé *tokenisation* est utilisé. Un *tokenizer* est un outil logiciel qui prend un texte en entrée et le découpe en une séquence de *tokens*, en identifiant les unités linguistiques significatives.

**EXEMPLE** **Tokenisation**

Un *tokenizer* pourrait transformer ces phrase en la séquence de *tokens* suivante :


Chaque élément, représenté par une couleur différente, est un token. Il est important de noter que les espaces sont également considérés comme des tokens, car ils font partie intégrante de la structure de la phrase.

Il est crucial de bien comprendre que, à partir de maintenant, lorsque nous parlerons de « mots » ou de « texte » dans le contexte des *LLM*, il s'agira en réalité de **séquences de *tokens*.** Toute la magie de la compréhension et de la génération du langage par les *LLM* opère au niveau de ces briques élémentaires.

## 1.2 L'architecture Transformer : l'orchestre de l'attention

Maintenant que nous avons compris ce que sont les *tokens*, plongeons-nous dans l'architecture qui propulse les *LLM* modernes : les *Transformers*. Ils constituent une innovation majeure en matière de réseaux de neurones profonds, spécialement conçus pour traiter le langage de manière efficace et puissante.

### *1.2.1 Une architecture révolutionnaire*

Imaginez le *Transformer* comme un **orchestre** complexe, où chaque instrument (chaque partie du réseau de neurones) joue un rôle spécifique, mais où l'harmonie globale (la compréhension et la génération du langage) émerge de l'interaction de toutes les parties.

La clé de la puissance des *Transformers* réside dans un mécanisme déterminant : l'**attention**. Contrairement aux anciens modèles de langage qui traitaient le texte de manière strictement séquentielle, mot après mot, les *Transformers* sont capables de regarder tous les *tokens* d'une phrase en même temps. C'est un peu comme lire une phrase entière d'un coup d'œil pour en saisir le contexte global avant de revenir sur chaque mot individuellement.

### *1.2.2 Le mécanisme d'attention : faire des liens entre les tokens*

L'attention permet au *Transformer* de calculer l'importance et la relation de chaque *token* avec tous les autres *tokens* de la phrase. Pour chaque *token*, le *Transformer* se pose en quelque sorte les questions suivantes :

- « Parmi tous les autres *tokens* de cette phrase, lesquels sont les plus importants pour comprendre le sens de ce *token*-ci ? »
- « Quels sont les liens (sémantiques, grammaticaux, contextuels) entre ce *token* et les autres *tokens* de la phrase ? »

Pour illustrer, reprenons notre exemple : « Le chat noir joue avec la pelote de laine car **il** est joueur. ». Concentrez-vous sur le *token* « **il** ». Pour comprendre à qui se réfère ce pronom, notre cerveau ne se contente pas de lire le mot « il » isolément. Nous faisons instinctivement le lien avec « **chat noir** », mentionné précédemment dans la phrase, en « portant attention » à chat noir pour interpréter correctement « il ».

C'est précisément ce que fait le mécanisme d'attention des *Transformers*, mais de manière mathématique et automatisée. Pour chaque *token* de la phrase, le modèle calcule des scores d'attention avec tous les autres *tokens*. Un score d'attention élevé entre deux *tokens* indique qu'ils sont fortement liés et que l'un est important pour comprendre l'autre. Dans notre exemple, le score d'attention entre « il » et « chat » sera élevé, car « chat » est essentiel pour déterminer à quoi se réfère « il ». En revanche, le score d'attention entre « il » et « joue » sera probablement plus faible : bien que « joue » apporte un contexte, ce mot n'est pas directement pertinent pour identifier qui est « il ».

En résumé, l'attention permet au *Transformer* de construire une compréhension **contextuelle** de chaque *token* en considérant l'ensemble de la phrase et les relations entre tous les *tokens*. C'est cette capacité à saisir le contexte global qui donne aux *Transformers* leur puissance et leur cohérence dans la génération de texte.

### 1.3 Les poids du modèle : la mémoire et les connaissances internes

Comment les *Transformers* **apprennent-ils** à réaliser ce mécanisme complexe d'attention et de compréhension du langage ? La réponse réside dans les paramètres internes du modèle, que l'on appelle souvent les **poids**. Imaginez les poids comme des **curseurs de réglage** à l'intérieur du *Transformer*. Ce sont des valeurs numériques qui sont ajustées automatiquement pendant la phase d'apprentissage du modèle. En modifiant ces poids, le modèle apprend à reconnaître des motifs dans le langage, à établir des relations entre les *tokens* et, finalement, à prédire le *token* suivant de manière pertinente.

#### *1.3.1 Lien entre les poids et les tokens*

Les poids représentent en quelque sorte la « mémoire » et les « connaissances » du modèle sur le langage. Ils codent par exemple :

- **Les relations statistiques entre les tokens :** Quels *tokens* ont tendance à apparaître ensemble ? Quels *tokens* se suivent fréquemment ? Par exemple, le modèle apprendra que « le » est souvent suivi par un nom commun, ou que « et » relie deux éléments.
- **Les relations sémantiques entre les tokens :** Quels *tokens* ont des significations similaires ? Quels *tokens* sont liés conceptuellement ? Par exemple, le modèle apprendra que « chat » et « félin » sont sémantiquement proches, ou que « manger » est relié à « nourriture ».

- **Les règles grammaticales et syntaxiques :** Bien que les *LLM* n'apprennent pas explicitement les règles de grammaire, ils les internalisent implicitement à travers les données d'entraînement. Les poids capturent des régularités syntaxiques comme l'ordre des mots, les accords, etc.

### 1.3.2 Processus d'apprentissage : pré-entraînement et affinage (fine-tuning)

L'entraînement des *LLM* se déroule généralement en deux phases principales :

- **Pré-entraînement :** Le modèle est d'abord exposé à d'énormes quantités de textes bruts sans étiquettes spécifiques (c'est-à-dire sans annotations manuelles ou catégorisations - contrairement à l'apprentissage supervisé où chaque exemple serait associé à une classification pré-établie). L'objectif est d'apprendre à prédire le *token* suivant dans une séquence de texte. Autrement dit, le modèle lit des phrases et tente de deviner chaque mot à venir. Il ajuste alors ses poids pour minimiser l'erreur de prédiction. On peut comparer cela à un enfant qui apprend à lire : au début, il fait beaucoup d'erreurs, mais à force d'exercice, il ajuste sa compréhension et devient de plus en plus précis. Pendant le pré-entraînement, le modèle acquiert ainsi une compréhension générale du langage et du monde (à travers tout ce qu'il lit).
- **Affinage (fine-tuning) :** Après le pré-entraînement, le modèle possède une vaste connaissance du langage. L'étape d'affinage consiste à spécialiser le modèle pour des tâches plus ciblées. On va le ré-entraîner sur des ensembles de données plus restreints, étiquetés pour des tâches particulières (par exemple la traduction, l'analyse de sentiments, le question-réponse, etc.). Cette étape ajuste les poids du modèle de manière plus fine pour optimiser ses performances dans ces tâches spécifiques. C'est comme si, après avoir appris les bases de la musique, un musicien décidait de se spécialiser dans le piano : il a déjà l'oreille musicale générale, mais il va maintenant se concentrer et s'entraîner intensivement sur son instrument pour devenir virtuose.

## 1.4 La prédiction du token suivant : le moteur de la génération de texte

Une fois le *Transformer* entraîné, avec ses poids ajustés et ses connaissances du langage internalisées, il est prêt à être utilisé pour **générer du texte**. C'est la phase d'inférence. La génération d'un texte à partir d'un *prompt* se fait de manière itérative, *token* par *token*.

### 1.4.1 Réception du prompt initial

Le modèle reçoit d'abord un *prompt*, c'est-à-dire un texte de départ fourni par l'utilisateur (par exemple une question, une consigne ou le début d'une phrase). Ce *prompt* est immédiatement converti en séquence de *tokens* par le *tokenizer*, pour que le modèle puisse le traiter.

### 1.4.2 Calcul des probabilités des tokens suivants

Le *Transformer* analyse le *prompt* (cette séquence de *tokens* d'entrée) en utilisant son mécanisme d'attention et l'ensemble de ses poids appris. À partir du contexte fourni par le *prompt*, il va calculer, pour **chaque *token* possible** qui pourrait suivre, une probabilité d'apparition. Cette probabilité représente la **vraisemblance** que ce *token* soit le meilleur candidat pour continuer le texte de manière

cohérente à ce moment-là. On peut imaginer qu'il construit un classement de tous les *tokens* envisageables, du plus probable au moins probable en fonction du contexte. Les *tokens* avec les probabilités les plus élevées sont ceux que le modèle considère comme les plus pertinents pour poursuivre la phrase et ainsi, construire sa réponse.

### 1.4.3 Sélection du prochain token

Parmi ces *tokens* possibles, le modèle doit en choisir un pour qu'il devienne le prochain *token* généré dans la réponse. Différentes stratégies de sélection peuvent être utilisées :

- **Choix du token le plus probable (décodage déterministe) :** sélectionner systématiquement le *token* ayant la probabilité la plus élevée. Cette approche conduit à des textes souvent cohérents et sûrs, mais qui peuvent paraître un peu prévisibles ou « plats » car le modèle ne prend aucun risque.
- **Échantillonnage aléatoire pondéré (décodage stochastique) :** tirer au sort le prochain *token* en respectant la distribution de probabilités calculée.

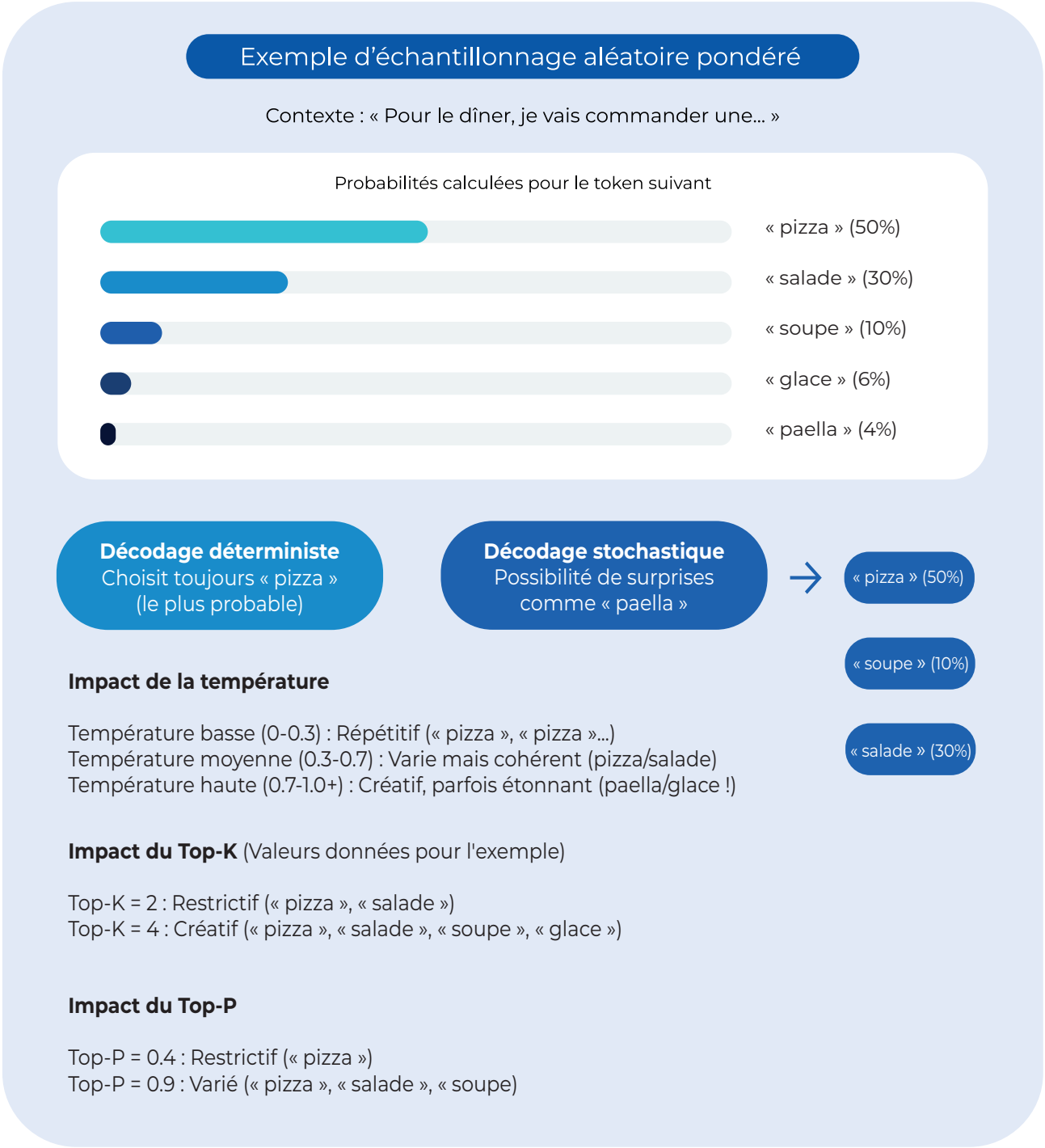
Ainsi, un token très probable a de fortes chances d'être choisi, mais un *token* moins probable peut aussi être sélectionné de temps en temps. Cette approche introduit plus de variété et de créativité dans la génération. Un paramètre appelé température contrôle ce degré de prise de risque :

- **Températures basses (0-0.3) :** Le modèle se comporte de façon presque déterministe, produisant des réponses très prévisibles et cohérentes. Idéal pour des tâches factuelles comme répondre à des questions précises ou générer du contenu technique.
- **Températures moyennes (0.3-0.7) :** Offre un équilibre entre créativité et cohérence, permettant une certaine variabilité tout en maintenant un fil conducteur logique.
- **Températures élevées (0.7-1.0 ou plus) :** Le modèle ose beaucoup plus de tokens moins probables, ce qui rend le texte plus original et créatif, mais augmente le risque d'incohérence ou d'hallucination (réponse qui semble plausible mais qui est factuellement incorrecte ou non fidèle à la réalité sur le fond).

- **Mécanismes de filtrage complémentaires :** D'autres paramètres peuvent affiner la sélection des tokens :
  - **Top-K :** Limite la sélection aux K *tokens* les plus probables. Des valeurs basses (K=20-40) produisent des réponses plus conservatrices et précises, tandis que des valeurs élevées (K>50) permettent plus de variété.
  - **Top-P (nucleus sampling) :** Ne considère que les *tokens* dont la probabilité cumulée atteint un seuil P. Des valeurs basses (0.5-0.7) sont plus restrictives, tandis que des valeurs élevées (0.9-1.0) offrent plus de liberté créative.
  - **Limite de tokens :** Définit simplement la longueur maximale de la réponse générée, servant de critère d'arrêt.

Selon la tâche à accomplir, différentes configurations sont recommandées. Par exemple, pour une réponse factuelle, une température basse (0.1) avec un top-K modéré (40) et un top-P de 0.95 donne

d'excellents résultats. Pour une génération créative comme une histoire ou un poème, privilégiez une température plus élevée (0.7-0.9) avec un top-K plus large (50) et un top-P proche de 1 (0.98).




- **Ajout du token et itération :** Une fois le *token* sélectionné selon ces paramètres, il est ajouté à la suite du texte déjà généré. Ce nouveau *token* fait désormais partie du contexte pris en compte. Le modèle recommence alors : il recalcule les probabilités pour le *token* suivant en considérant ce contexte enrichi, puis en choisit un, et ainsi de suite.

Ce processus itératif se poursuit jusqu'à ce que le modèle atteigne la longueur de texte désirée, ou jusqu'à ce qu'il génère un *token* spécial de fin (par exemple un symbole marquant la fin de la réponse). C'est cette boucle de prédiction *token* par *token*, guidée par l'attention et les poids du *Transformer*, qui est au cœur de la génération de texte des *LLM*. À chaque étape, le mot suivant s'appuie sur le contexte des mots précédents, ce qui permet de construire des textes souvent étonnamment cohérents, pertinents et même créatifs.

### 1.5 Les contextes longs : la « mémoire » des LLM modernes

Les modèles de langage récents ont réalisé des progrès spectaculaires dans leur capacité à gérer des **contextes très longs**. Alors que les premiers *LLM* avaient une « mémoire » limitée et pouvaient rapidement oublier le début d'une conversation ou d'un texte long, les modèles modernes peuvent traiter des contextes de plusieurs **dizaines de milliers de tokens**.

**La fenêtre de contexte :** c'est un concept-clé pour comprendre cette capacité et ses limites. La fenêtre de contexte désigne la quantité maximale de texte (mesurée en *tokens*) que le modèle peut prendre en compte à un moment donné pour générer une réponse. En d'autres termes, cela correspond un peu à la mémoire à court terme du modèle – ce qu'il peut « voir » du prompt ou de la conversation en cours. Si un prompt dépasse cette limite de longueur, les informations les plus anciennes sont généralement ignorées, ce qui peut nuire à la pertinence ou à la cohérence des réponses.

Malgré cette amélioration, il existe des défis. Des études ont mis en évidence le phénomène du « *lost in the middle* » (**perdu au milieu**), qui indique que les modèles ont parfois du mal à traiter correctement les informations situées au milieu d'un très long contexte : ils tendent à mieux se souvenir du début et de la fin, aux dépens du milieu. Les éléments initiaux ou les plus récents d'un prompt étendu peuvent ainsi avoir plus de poids dans la réponse que des éléments importants enfouis en plein milieu du contexte.

**La capacité à gérer de longs contextes est cruciale pour :**

- **Des conversations plus cohérentes et naturelles :** Le modèle peut se souvenir des échanges précédents dans une discussion et s'y référer, évitant de poser plusieurs fois la même question ou de contredire ce qui a été dit.
- **La compréhension de documents longs :** Le modèle peut ingérer de longs articles, rapports ou histoires et répondre à des questions en ayant gardé le fil conducteur et la cohérence d'ensemble.
- **Des tâches complexes nécessitant une mémoire étendue :** Par exemple, pour générer un récit cohérent de plusieurs pages, ou pour résoudre un problème en plusieurs étapes où il faut garder trace d'informations introduites tôt dans l'énoncé.

Des améliorations architecturales et des techniques d'entraînement ont permis d'étendre considérablement cette fenêtre de contexte. En optimisant le mécanisme d'attention, les *Transformers* modernes parviennent à conserver une forme de « mémoire » sur les *tokens* précédents, rendant la génération plus cohérente sur la durée. Toutefois, les recherches continuent pour atténuer le problème du *lost in the middle* et, plus globalement, pour améliorer encore la gestion des très longs contextes par les *LLM*.

En résumé, dans ce chapitre nous avons exploré en profondeur le fonctionnement interne des modèles de langage génératifs *(LLM)*. Nous avons découvert que :

- Les *LLM* travaillent avec des *tokens*, les briques élémentaires du langage.
- L'architecture *Transformer*, via son mécanisme d'attention, relie les *tokens* entre eux pour comprendre le contexte global d'une phrase.
- Les poids du modèle, ajustés lors de l'entraînement, encodent les connaissances linguistiques et factuelles, ainsi que les régularités du langage.
- La génération de texte s'effectue de manière itérative en prédisant *token* après token en fonction du contexte et des probabilités calculées.
- Les *LLM* modernes gèrent des contextes très longs grâce à une fenêtre de contexte élargie, offrant une meilleure « mémoire » de la conversation, malgré des défis persistants (comme la tendance à oublier le milieu d'un long texte).

Cette compréhension des mécanismes internes des *LLM* (*tokens*, attention, mémoire, etc.) est essentielle pour aborder le *prompt engineering* de manière efficace. En saisissant comment les modèles « pensent » et produisent du texte, nous pourrons concevoir des prompts plus judicieux afin d'obtenir les résultats souhaités.

Dans le chapitre suivant, nous allons justement explorer les principes de base du *prompt engineering* et découvrir comment utiliser ces connaissances pour formuler des instructions claires et précises, afin de tirer le meilleur parti des *LLM*. Préparez-vous à entrer dans le vif du sujet et à découvrir les premières techniques pour maîtriser l'art du *prompt* !

# /Chapitre 2
# Principes de base pour rédiger un prompt efficace

Maintenant que nous avons exploré en profondeur le fonctionnement interne des modèles de langage, il est temps de passer à la pratique et de nous intéresser à l'art du *prompt engineering*. Si les *LLM* sont des outils puissants, leur efficacité dépendra grandement de notre capacité à leur fournir des instructions claires, précises et pertinentes. Ce chapitre est dédié aux principes de base du *prompt engineering* – ces **notions fondamentales** qui vous guideront dans la création de *prompts* efficaces et performants.

Pour vous aider à mémoriser ces principes, vous allez bientôt comprendre pourquoi (CCRFE) « **Ce Chat Rigole Fort, Evidemment !** ». Nous allons explorer les cinq piliers essentiels du *prompt engineering* : **Clarté**, **Contexte**, **Rôle**, **Format**, et **Exemples**. Ces éléments sont la clé pour formuler des requêtes efficaces et obtenir des résultats de qualité. Préparez-vous à découvrir comment chacun de ces principes contribue à rendre vos interactions avec les modèles de langage plus précises, pertinentes et puissantes.

## 2.1 Les principes clés d'un prompt performant

Au-delà des différents types de *prompts* que nous aborderons ensuite, il existe des principes fondamentaux qui sous-tendent la conception d'un *prompt* réussi.

### Principe 1 : Clarté et précision – Parler le langage de l'IA

- **L'importance de la clarté** – Les *LLM*, bien que sophistiqués, interprètent vos prompts de manière littérale. Un *prompt* vague, ambigu ou mal formulé risque d'être mal compris et de produire des résultats décevants. La clarté est donc primordiale.
- **La précision est votre alliée** – Soyez précis dans vos instructions. Évitez les formulations générales et approximatives. Plus vous serez explicite sur ce que vous attendez, plus le modèle sera en mesure de vous fournir une réponse pertinente.
- **Utilisez un langage simple et direct** – N'essayez pas d'être trop littéraire ou poétique dans vos *prompts* (du moins pas au début). Privilégiez un langage accessible, direct et non ambigu. En quelque sorte, parlez le langage de l'IA, un langage fondé sur la logique et la déduction.
- **Décomposez les tâches complexes** – Si votre demande est complexe, n'hésitez pas à la découper en étapes plus simples et à les exprimer clairement dans votre *prompt*. (Nous verrons plus loin des techniques avancées pour aider le modèle à gérer étape par étape les problèmes compliqués.)

**EXEMPLES**

- **Prompt imprécis :** « *Parle-moi des chats.* » (Trop vague : le modèle ne sait pas quel aspect des chats vous intéresse.)
- **Prompt clair et précis :** « Décris les principales caractéristiques physiques et comportementales des chats domestiques, en 3 paragraphes maximum. » (Beaucoup plus clair sur ce qui est attendu.)

**Principe 2 : Contexte riche et pertinent – Guider l'attention du modèle**

- **Le contexte est roi** – Comme nous l'avons vu au Chapitre 1, les *Transformers* excellent à comprendre le contexte grâce au mécanisme d'attention. Fournir un contexte riche et pertinent est essentiel pour guider l'attention du modèle et l'orienter vers la réponse souhaitée.
- **Définissez le cadre** – Situez clairement le contexte de votre demande. De quel sujet parle-t-on ? Quel est le but de la génération ? Quelles sont les informations importantes à prendre en compte ?
- **Fournissez les informations nécessaires** – N'hésitez pas à inclure dans votre *prompt* toutes les informations que vous jugez pertinentes pour que le modèle comprenne bien votre demande et puisse y répondre efficacement. Cela peut inclure des mots-clés, des précisions sur la situation, des données spécifiques, etc.
- **Adaptez le contexte à la tâche** – Le niveau de détail du contexte dépendra de la complexité de la tâche. Pour une question simple, un contexte minimal peut suffire. Pour un problème complexe, un contexte riche et détaillé sera souvent crucial.

**EXEMPLES**

- **Prompt sans contexte :** « *Écris un poème.* » (Le modèle peut générer n'importe quel type de poème, sans direction.)
- **Prompt avec contexte :** « *Écris un court poème de style haïku sur le thème de la nature au printemps, en utilisant des images visuelles et des mots simples.* » (Le contexte guide le modèle vers un type de poème précis et un thème défini.)

**Principe 3 : Rôle ou un persona – Exploiter les connaissances spécialisées**

- **Attribuer un rôle au modèle** – Les *LLM* ont été entraînés sur d'énormes quantités de textes provenant de domaines très variés. Ils possèdent donc une vaste base de connaissances, parfois dispersée. Attribuer un rôle ou un persona au modèle dans votre *prompt* permet de le focaliser sur un domaine de connaissances spécifique et d'adopter un ton ou un style approprié.
- **Exploiter les connaissances spécialisées** – En définissant un rôle (expert médical, spécialiste en histoire, personnage fictif, etc.), vous encouragez le modèle à puiser dans les informations les plus pertinentes pour la tâche demandée, comme s'il mobilisait la partie de ses « connaissances » liée à ce rôle.
- **Influencer le style et le ton** – Le rôle attribué influence également le style d'écriture, le vocabulaire et le ton de la réponse. Par exemple, si vous demandez au modèle de répondre comme un professeur d'université, il produira un texte plus formel et académique que s'il répond comme un ami expliquant un concept simplement.

**EXEMPLES**

- **Prompt sans rôle défini :** « *Explique la théorie de la relativité.* » (Réponse potentiellement très technique et aride.)
- **Prompt avec rôle :** « *Imagine que tu es un professeur de physique expliquant la théorie de la relativité à des lycéens. Explique-leur les concepts clés de manière simple et imagée, en utilisant des analogies.* » (Réponse plus pédagogique, accessible et imagée, car le modèle adopte le persona professeur pédagogue.)

**Principe 4 : Format de sortie – Guider la présentation des informations**

- **Spécifiez le format souhaité** – Pour obtenir des réponses bien structurées et faciles à exploiter, il est important de spécifier clairement le format de sortie attendu dans votre *prompt*. Ne laissez pas le modèle choisir le format par défaut, prenez le contrôle !
- **Utilisez des indications de format** – Vous pouvez employer différentes techniques pour indiquer le format voulu :
  - Des instructions textuelles explicites dans le *prompt* (par ex. « *Réponds sous forme de liste à puces.* », « *Présente le résultat dans un tableau.* », « *Rédige un e-mail formel.* »).
  - Des exemples de format dans le *prompt* (montrer au modèle un modèle de réponse formatée, qu'il imitera).
  - Des balises ou du pseudo-code de formatage (JSON, XML, Markdown, etc.) dans le *prompt* pour forcer un format précis.
- **Limiter la longueur** – Si vous souhaitez une réponse concise ou d'une taille donnée, spécifiez une longueur maximale (nombre de mots, de phrases, de caractères). Cela évite les réponses trop longues ou hors-sujet.

**EXEMPLES**

- **Prompt sans indication de format :** « *Quels sont les avantages et les inconvénients des énergies renouvelables ?* » (Le modèle pourrait répondre dans un unique paragraphe très dense.)
- **Prompt avec indication de format :** « *Liste les avantages et les inconvénients des énergies renouvelables sous forme de deux listes à puces distinctes (avantages d'un côté, inconvénients de l'autre).* » (Le modèle fournira une réponse bien structurée en deux listes, facile à lire.)

**Principe 5 : Exemples pertinents – Montrer plutôt que dire**

- **Des exemples comme guides** – Les exemples sont des outils extrêmement efficaces pour guider les *LLM*. « « Un bon exemple vaut mieux qu'un long discours », dit l'adage, et il s'applique parfaitement au *prompt engineering*. Plutôt que de donner de longues explications abstraites, montrez au modèle ce que vous attendez en lui fournissant un ou plusieurs exemples concrets.

■ **Types d'exemples** – Vous pouvez utiliser plusieurs sortes d'exemples dans un *prompt* :

- Exemple de résultat souhaité – montrez au modèle ce à quoi devrait ressembler la réponse (par exemple, fournir une question et la réponse attendue correspondante).
- Exemple de style ou de ton – fournissez un extrait de texte illustrant le style ou le ton que vous attendez dans la réponse.
- Exemple de format – montrez un format de sortie particulier (par exemple, comment structurer une réponse sous forme de tableau ou de JSON).
- Few-shot learning – l'apprentissage par quelques exemples *(few-shot learning)* consiste à inclure dans le *prompt* plusieurs exemples complets (entrées et sorties attendues). Le modèle va ainsi apprendre par imitation à résoudre la nouvelle tâche en suivant le modèle des exemples. (Nous approfondirons cette technique de *few-shot* un peu plus loin dans ce chapitre.)

**EXEMPLES**

- **Prompt sans exemple :** « *Écris un message marketing pour une nouvelle application de méditation.* « (Le modèle pourrait générer un contenu générique sans style distinctif.)
- **Prompt avec exemples :** « *Crée un message marketing pour une nouvelle application de méditation appelée « ZenMind » en t'inspirant de ces exemples :*

*Exemple de résultat souhaité :*

*« Découvrez Fitness+, l'application qui transforme votre routine d'exercice en expérience personnalisée avec des entraîneurs experts. »*

*Exemple de style/ton :*

« *Imaginez un monde où chaque respiration vous rapproche de votre véritable potentiel… »*

*Exemple de format :*

*TITRE ACCROCHEUR*

*Phrase d'introduction émotionnelle*

*3 bénéfices-clés en puces*

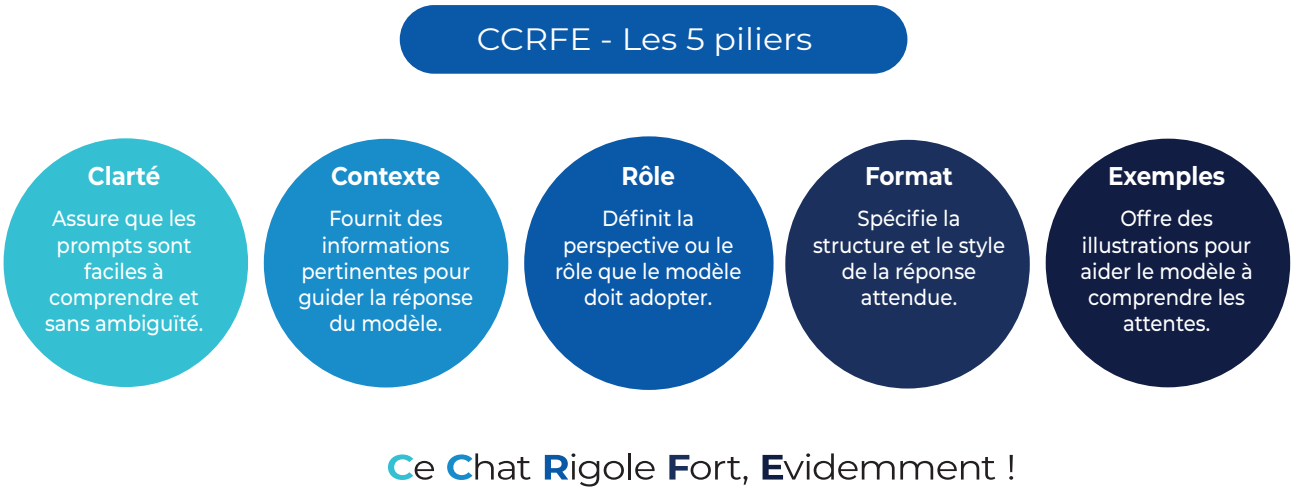
*Appel à l'action*

*Few-shot learning :*

*Pour application de sommeil : « Dormez mieux. Réveillez-vous revigoré. SleepWell transforme vos nuits avec des histoires apaisantes et des sons relaxants. »*

Remarque complémentaire :

Il est important de noter que la réussite d'un *prompt* ne réside pas uniquement dans la clarté, le contexte ou le format attendu. La capacité intrinsèque du modèle – déterminée par sa taille et la richesse de son corpus d'entraînement – joue un rôle crucial. Pour les modèles de grande taille, une légère imprécision dans le prompt pourra être corrigée grâce à leur vaste base de connaissances. À l'inverse, des modèles plus légers – comme par exemple le modèle Microsoft Phi 3 – disposent d'un corpus d'entraînement plus restreint. Pour ces derniers, il est indispensable d'adopter une approche très structurée : décomposer la tâche en sous-étapes, fournir un contexte détaillé et, de préférence, interagir en anglais. Cette variation de performance souligne l'importance d'adapter nos stratégies de formulation des *prompts* en fonction du modèle utilisé. Une telle adaptation permet de compenser les limitations inhérentes aux modèles plus légers et d'obtenir des réponses aussi précises que possible.




**2.2 Les différents types de prompts : des outils adaptés à chaque situation**

Maintenant que nous avons posé les principes fondamentaux, explorons les **différents types de *prompts*** que vous pouvez utiliser en pratique. Le choix du type de *prompt* dépendra de la tâche à réaliser, du niveau de contrôle que vous souhaitez exercer sur le modèle, et du contexte spécifique de votre application.

Voici les principales catégories de *prompts*, du plus simple au plus élaboré :

**2.2.1 La demande simple (instruction directe) : l'essentiel efficace**

- **Définition :** Le *prompt* le plus basique, consistant en une instruction claire et directe demandant au modèle de réaliser une tâche spécifique. C'est souvent suffisant pour des demandes simples et bien définies.
- **Caractéristiques :**
  - *Prompt* concis et facile à comprendre.
  - Va droit au but, sans fioritures.
  - Contient des instructions explicites sur ce qui est attendu.
- **Quand l'utiliser :**
  - Tâches simples et courantes (résumer un court texte, traduire une phrase, générer une brève description, etc.).
  - Lorsque vous savez précisément ce que vous voulez et qu'il n'est pas nécessaire de guider le modèle avec du contexte supplémentaire.

**EXEMPLE**

« *Écris un article de blog de 500 mots sur les bienfaits du jardinage pour la santé mentale.* »
(Ici le *prompt* est une simple instruction directe, clairement formulée.)

**2.2.2 Le prompt contextuel (avec fourniture de contexte) : orienter et préciser**

- **Définition :** Un *prompt* qui enrichit la demande simple en fournissant un contexte supplémentaire pour guider le modèle. Le contexte peut inclure des informations sur le sujet, le but de la tâche, le public cible, etc.

■ **Caractéristiques :**

- Inclut une instruction claire + un contexte pertinent.
- Aide le modèle à mieux comprendre la demande en lui donnant un cadre ou des détails additionnels.
- Permet de nuancer et préciser la tâche.

■ **Quand l'utiliser :**

- Tâches qui bénéficient d'une mise en contexte ou d'un cadre de référence.
- Lorsque vous voulez vous assurer que le modèle prend en compte certains éléments spécifiques dans sa réponse (par ex. le public cible, le style attendu, des détails de situation).

**EXEMPLE**

« *Imagine que tu es un guide touristique à Paris. Un touriste te demande : « Quels sont les monuments incontournables à visiter à Paris en une journée ? » Formule une réponse concise et pratique, en tenant compte que le touriste n'a qu'une journée sur place.* »
(Ici, en plus de la question, on donne du contexte : le modèle doit se mettre dans la peau d'un guide touristique et considérer la contrainte de temps.)

### 2.2.3 Le prompt instructif (instructions détaillées) : pour un contrôle précis

■ **Définition :** Un *prompt* qui fournit des instructions **très détaillées** et précises sur la tâche à réaliser, souvent avec plusieurs contraintes ou directives spécifiques. Ce type de *prompt* permet un contrôle plus fin sur la génération.

■ **Caractéristiques :**

- Donne des instructions étape par étape, ou une liste de critères très spécifiques à respecter.
- Permet de définir clairement des contraintes (longueur, format, style, éléments à inclure/éviter, etc.).
- Offre un haut niveau de contrôle sur la réponse du modèle.

■ **Quand l'utiliser :**

- Tâches complexes qui nécessitent des directives précises.
- Lorsque vous avez des exigences spécifiques en termes de format, de style, de contenu, etc., et que vous voulez contraindre la sortie du modèle à les respecter.
- Pour des tâches où la précision et la conformité à des règles sont importantes (par ex. générer un code dans un format précis, respecter une mise en forme réglementaire, etc.).

**EXEMPLE**

« *Rédige une courte histoire de science-fiction se déroulant sur une station spatiale en orbite autour de la Terre. Le personnage principal doit être un robot jardinier. L'histoire fera environ 800 mots et aura une fin ouverte qui laisse imaginer une suite. Utilise un style narratif à la troisième personne et un ton à la fois poétique et mélancolique.* »
(Ce *prompt* instructif donne de nombreux détails et contraintes : cadre, personnage, longueur ~800 mots, type de fin, style narratif, ton. Le modèle est ainsi très guidé dans sa génération.)

### 2.2.4 Le prompt few-shot (apprentissage par quelques exemples) : apprendre en montrant

- **Définition :** Un *prompt* qui inclut quelques **exemples complets** de la tâche souhaitée (paires entrée→sortie) pour guider le modèle. Le *few-shot learning* est une technique puissante où le modèle apprend à réaliser la tâche en imitant les exemples plutôt qu'en suivant uniquement des instructions abstraites.
- **Caractéristiques :**
  - Contient une instruction claire + quelques exemples d'entrée et leur sortie attendue.
  - Le modèle apprend du style, du format et du contenu de ces exemples.
  - Particulièrement efficace pour induire un style, un ton ou un format précis, ou pour des tâches difficiles à décrire par de simples instructions.
- **Quand l'utiliser :**
  - Tâches pour lesquelles il est compliqué de décrire les critères de réussite, mais facile de montrer des exemples de bonnes réponses.
  - Pour orienter le modèle vers un style/ton/formalisme particulier (fournir un ou plusieurs exemples conformes au style souhaité).
  - Pour des tâches où l'on veut que le modèle imite un certain comportement ou reproduise des motifs (par exemple, parler comme Shakespeare – montrer un extrait de texte Shakespearien).

**EXEMPLE**

**Exemple :**

*« Rédige la description du produit « Beurre de Karité » pour un site e-commerce de cosmétiques naturels, en suivant ce modèle :*

***Huile d'argan bio***

*Extrait des arbres d'argan du Maroc, cette huile précieuse est un véritable élixir de beauté. Riche en antioxydants et en vitamine E, elle nourrit intensément la peau et les cheveux. Son absorption rapide laisse la peau douce sans effet gras. Idéale comme sérum quotidien ou en masque capillaire hebdomadaire.*

***Masque à l'argile verte***

*Notre argile verte, récoltée dans les montagnes françaises, purifie et détoxifie la peau en profondeur. Ce masque absorbe l'excès de sébum, resserre les pores et calme les inflammations. Sa formule 100% naturelle convient parfaitement aux peaux mixtes à grasses cherchant un teint clarifié et mat.* »

(Cet exemple fournit deux descriptions de produits cosmétiques qui suivent une structure claire : nom du produit en gras, description détaillant l'origine, les bénéfices principaux, le ressenti lors de l'utilisation, et les conseils d'application. Chaque description maintient un ton élégant et évocateur tout en restant informative. Le modèle peut ainsi générer une troisième description pour le beurre de karité en respectant le même style, la même longueur et la même structure.)

### 2.2.5 Le prompt « données brutes » (data-driven) : analyser et transformer l'information

- **Définition :** Un *prompt* qui fournit des **données brutes** au modèle et lui demande de les analyser, de les transformer ou d'en extraire des informations. C'est utile pour des tâches d'analyse de données textuelles, de classification, de résumé, etc., où le *prompt* sert à encoder les données à traiter suivies d'une consigne de traitement.
- **Caractéristiques :**
  - Fournit explicitement des données en entrée (liste, texte brut, tableau, etc.) intégrées au *prompt*.
  - Contient ensuite une instruction claire sur ce qu'il faut faire avec ces données (analyser, classer, résumer, extraire une info spécifique, etc.).
  - Le modèle agit alors comme un outil de traitement et d'analyse sur les données textuelles fournies, plutôt que de puiser dans son savoir entraîné uniquement.
- **Quand l'utiliser :**
  - Analyse de sentiments de textes (fournir une liste de commentaires clients par ex.).
  - Classification de documents ou de courts textes (spam vs non-spam, avis positifs vs négatifs, etc.).
  - Résumé de longs textes ou conversations (fournir le texte puis demander un résumé).
  - Extraction d'informations spécifiques d'un document (donner un texte brut et poser une question pointue sur une info dans ce texte).

**EXEMPLE**

« *Voici une liste de commentaires clients sur notre nouveau smartphone :*
« *Super téléphone, très rapide et bel écran.* »
« *Batterie un peu faible...* »
« *Excellent rapport qualité/prix !* »
*[...]*
*Analyse ces commentaires et dis-moi pour chaque commentaire s'il est globalement positif, négatif ou neutre. Présente le résultat sous forme de liste avec chaque commentaire suivi de sa classification (positif, négatif ou neutre).* »
(Dans cet exemple, le *prompt* fournit une liste brute de commentaires clients puis demande au modèle de les analyser chacun. Le modèle doit renvoyer, pour chaque commentaire, une catégorisation du sentiment. On lui a bien spécifié le format de sortie attendu – une liste commentée.)

**En conclusion de ce chapitre**, le *prompt engineering* est un domaine en constante évolution. Depuis les premiers *prompts* très simples, nous avons vu émerger des techniques de plus en plus sophistiquées pour interagir avec les *LLM* et exploiter leur potentiel. Dans ce chapitre, nous avons exploré les principes de base – pour créer des *prompts* efficaces : la clarté et la précision, un contexte riche, un rôle défini, le contrôle du format, et l'utilisation d'exemples. Nous avons également passé en revue les différents types de prompts, des demandes simples aux prompts orientés données en passant par le *few-shot learning*, afin de vous offrir une boîte à outils variée et adaptable à différentes situations.

En maîtrisant ces principes de base et en sachant choisir le type de *prompt* approprié à chaque besoin, vous poserez des bases solides pour exploiter pleinement le potentiel des modèles de langage. Les chapitres suivants de ce livre blanc vont justement **approfondir ces techniques** et vous présenter des méthodes plus **avancées** pour affiner vos *prompts* et obtenir des résultats encore plus impressionnants. Préparez-vous à devenir un véritable expert en *prompt engineering* !

## Tableau synthétique des types de prompts

| Type | Définition | Caractéristiques | Cas d'usage | Exemple |
|---|---|---|---|---|
| Demande simple | Instruction directe | Concis<br>Explicite | Tâches simples<br>Besoins clairs | « Résume en 3 points les avantages du télétravail. » |
| Contextuel | Prompt + contexte | Cadre précis<br>Nuancé | Réponses ciblées<br>Public spécifique | « En tant que nutritionniste expliquant à un athlète, décris les bienfaits des protéines après l'entraînement .» |
| Instructif | Directives détaillées | Contraintes précises<br>Contrôle élevé | Tâches complexes<br>Format strict | « Crée un email commercial pour une offre de -30% sur des chaussures de sport : max 150 mots, ton enthousiaste, inclure 3 bénéfices et un appel à l'action .» |
| Few-shot | Exemples + instruction | Apprentissage par imitation<br>Modèles à suivre | Style spécifique<br>Format difficile à décrire | « Décris cet accessoire de sport comme les exemples :<br>[Ex.1] [Ex.2 ]» |
| Data-driven | Données + consigne | Analyse de contenu<br>Traitement ciblé | Classification<br>Extraction d'infos | « Voici les résultats mensuels de vente [données jan-déc]. Identifie les 3 mois les plus performants et explique les tendances possibles. » |

# /Chapitre 3
# Raisonnement avancé – Penser étape par étape avec la Chaîne de Pensée

Dans les chapitres précédents, nous avons appris à formuler des *prompts* clairs et structurés, et même à fournir des exemples pour guider le modèle. Cependant, face à des problèmes **complexes** ou des questions qui nécessitent de la réflexion en plusieurs étapes, un *prompt* basique, même bien formulé, peut ne pas suffire. C'est ici qu'interviennent des techniques avancées de **décomposition du raisonnement** pour aider le *LLM* à mieux raisonner.

L'idée générale est souvent de **décomposer** un problème complexe en étapes plus simples, ou d'encourager le modèle à **expliciter** ses étapes de réflexion au lieu de donner directement une réponse. Dans ce chapitre, nous explorons la technique de la **Chaîne de Pensée** *(Chain-of-Thought, CoT)*, qui est l'une des percées majeures pour améliorer les capacités de raisonnement des *LLM*. Nous verrons comment inciter le modèle à réfléchir « tout haut » et comment cette approche peut être combinée avec d'autres astuces comme l'**auto-cohérence** pour fiabiliser encore plus les réponses.

### 3.1 La technique de la Chaîne de Pensée (CoT)

La **Chaîne de Pensée** *(Chain-of-Thought)* consiste à demander au modèle de fournir une **explication pas à pas** de son raisonnement avant de donner sa réponse finale. Plutôt que de produire directement la réponse, le *LLM* génère d'abord une suite logique d'étapes intermédiaires, un peu comme un brouillon, puis conclut avec la solution. L'idée est de mimer le processus de pensée humain : face à un problème difficile, nous avons tendance à le décomposer en sous-problèmes ou à dérouler une réflexion étape par étape pour y répondre.

**Comment mettre en œuvre une chaîne de pensée ?** Il existe deux approches courantes :

- **Few-shot CoT :** Fournir au modèle un *prompt* contenant quelques exemples comportant chacun la question, la chaîne de pensée humaine détaillée menant à la réponse, et la réponse finale. Par imitation, le modèle va apprendre à produire sa propre chaîne de pensée pour une nouvelle question.
- **Zero-shot CoT :** Fournir une simple instruction additionnelle du type « Réfléchissons étape par étape » *("Let's think step by step »)*, éventuellement suivie d'un début de réflexion, pour pousser le modèle à dérouler un raisonnement avant d'aboutir à la réponse. Étonnamment, cette petite phrase peut suffire à activer le mode « raisonnement séquentiel » dans les grands modèles récents.

**Pourquoi la chaîne de pensée améliore-t-elle les résultats ?** L'explication est plus statistique que magique. Durant son entraînement, un *LLM* a rarement rencontré des problèmes complexes identiques à ceux qu'on lui pose, rendant peu probable qu'il ait mémorisé directement leurs solutions. En revanche, il a été exposé à de nombreux exemples de décomposition de problèmes et de raisonnements intermédiaires. Lorsqu'on l'incite à expliciter des étapes, on l'oriente vers ces patterns de décomposition qu'il connaît bien.

Chaque sous-problème ainsi créé a une probabilité bien plus élevée de correspondre à des patterns sur lesquels le modèle a été entraîné. De nombreuses études ont montré que le *CoT* permet des améliorations spectaculaires des performances sur des tâches nécessitant un raisonnement complexe, précisément parce qu'il transforme un problème statistiquement rare en une série de sous-problèmes statistiquement plus fréquents dans les données d'entraînement.

**Un avantage supplémentaire du CoT** est qu'il rend la réponse du modèle plus interprétable : en observant la chaîne d'étapes générée, l'utilisateur peut suivre la progression logique et identifier plus facilement les erreurs éventuelles dans la séquence de traitement.

**Limites :** Il est important de comprendre que la chaîne de pensée n'est pas une solution miracle. Même si la décomposition du problème semble logique et bien structurée, cela ne garantit pas que chaque étape soit correcte. En effet, si le modèle rencontre un sous-problème pour lequel il n'a pas de pattern statistique fiable dans son entraînement, il risque simplement d'inventer une solution qui paraît plausible. Le raisonnement peut sembler cohérent tout en comportant des erreurs fondamentales à certaines étapes.

De plus, le fait que *CoT* fonctionne mieux avec les très grands modèles n'est pas mystérieux: ces modèles ont simplement été exposés à un volume bien plus important de patterns de raisonnement et de sous-problèmes durant leur entraînement. Ils ont donc une « bibliothèque » de patterns plus riche pour chaque étape de décomposition. Les petits modèles, ayant vu moins d'exemples, ont plus de « trous » dans leur couverture de patterns, ce qui explique pourquoi *CoT* peut parfois dégrader leurs performances au lieu de les améliorer.

Cette observation soulève d'ailleurs un défi majeur pour l'avenir: la nécessité d'une meilleure transparence sur ce que ces modèles ont réellement appris et quels types de patterns ils maîtrisent véritablement. Sans cette connaissance, il reste difficile de prédire sur quels problèmes un modèle donné sera fiable, même avec une décomposition *CoT*.

Malgré ces réserves, la Chaîne de Pensée est un outil précieux pour tout *prompt engineer* confronté à des problèmes complexes. Voyons un exemple concret pour illustrer son utilisation.

**EXEMPLE**

- **Avec Chatgpt 4o (Sans instruction de CoT) :**

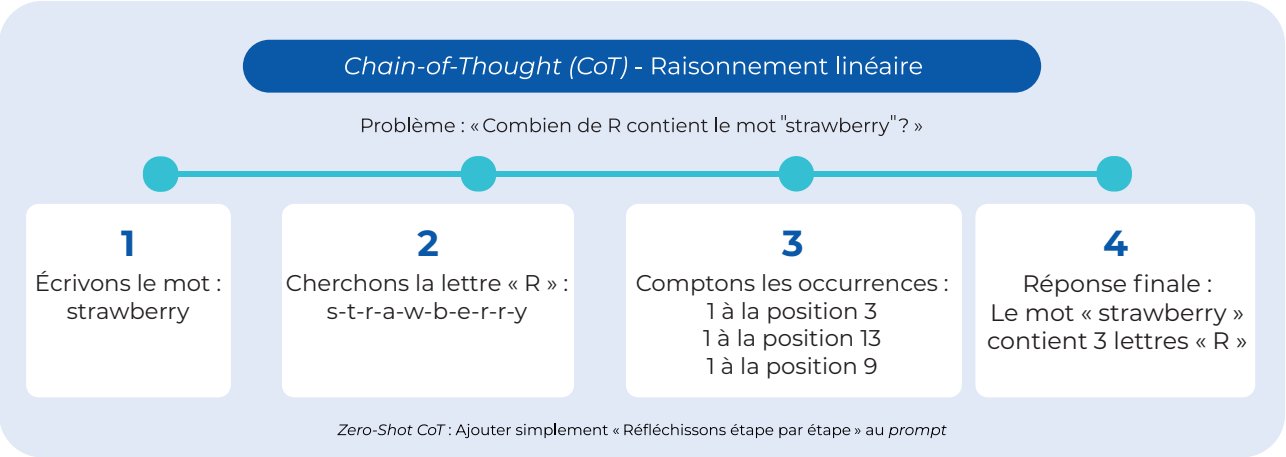
  **Question :** Combien de R contient le mot « *strawberry* » ?

  **Réponse :** Le mot « *strawberry* » contient deux « R ».

- **Avec Chatgpt 4o (Avec instruction de CoT) :**

  **Question :** Combien de R contient le mot « *strawberry* » ? Tu vas raisonner étape par étape.

  **Réponse :** D'accord, procédons étape par étape pour déterminer combien de « R » contient le mot « *strawberry* » :




En résumé, la Chaîne de Pensée incite le *LLM* à **raisonner de manière structurée** plutôt qu'à répondre d'un seul coup. C'est une technique particulièrement indiquée pour : les problèmes de mathématiques en langage naturel, les questions de logique ou de bon sens complexes, les raisonnements symboliques (manipuler des formules ou des symboles), et plus généralement toute tâche qu'on peut décomposer en une séquence d'étapes logiques.

### 3.2 L'auto-cohérence (Self-Consistency) : fiabiliser la chaîne de pensée

Même avec une bonne chaîne de pensée, le modèle peut parfois emprunter un chemin de raisonnement erroné qui le mène à une mauvaise réponse. Et il n'est pas toujours facile de déterminer, à la seule lecture de la chaîne, si la conclusion est correcte. C'est là qu'intervient la technique de l'**auto-cohérence** *(Self-Consistency)*, qui vise à améliorer la fiabilité du résultat en faisant appel à un consensus parmi plusieurs raisonnements.

#### 3.2.1 Principe

Plutôt que de faire confiance à **une seule** chaîne de pensée, l'idée est de générer **plusieurs chaînes de pensée indépendantes** pour la même question, puis de **faire voter** ces réponses entre elles.

En pratique, on exécute le modèle plusieurs fois. On obtient ainsi par exemple 5 raisonnements complets (parfois différents les uns des autres) menant possiblement à 5 réponses finales. Ensuite, on regarde quelle réponse revient le plus fréquemment parmi ces tentatives et on la choisit comme réponse finale par vote majoritaire.

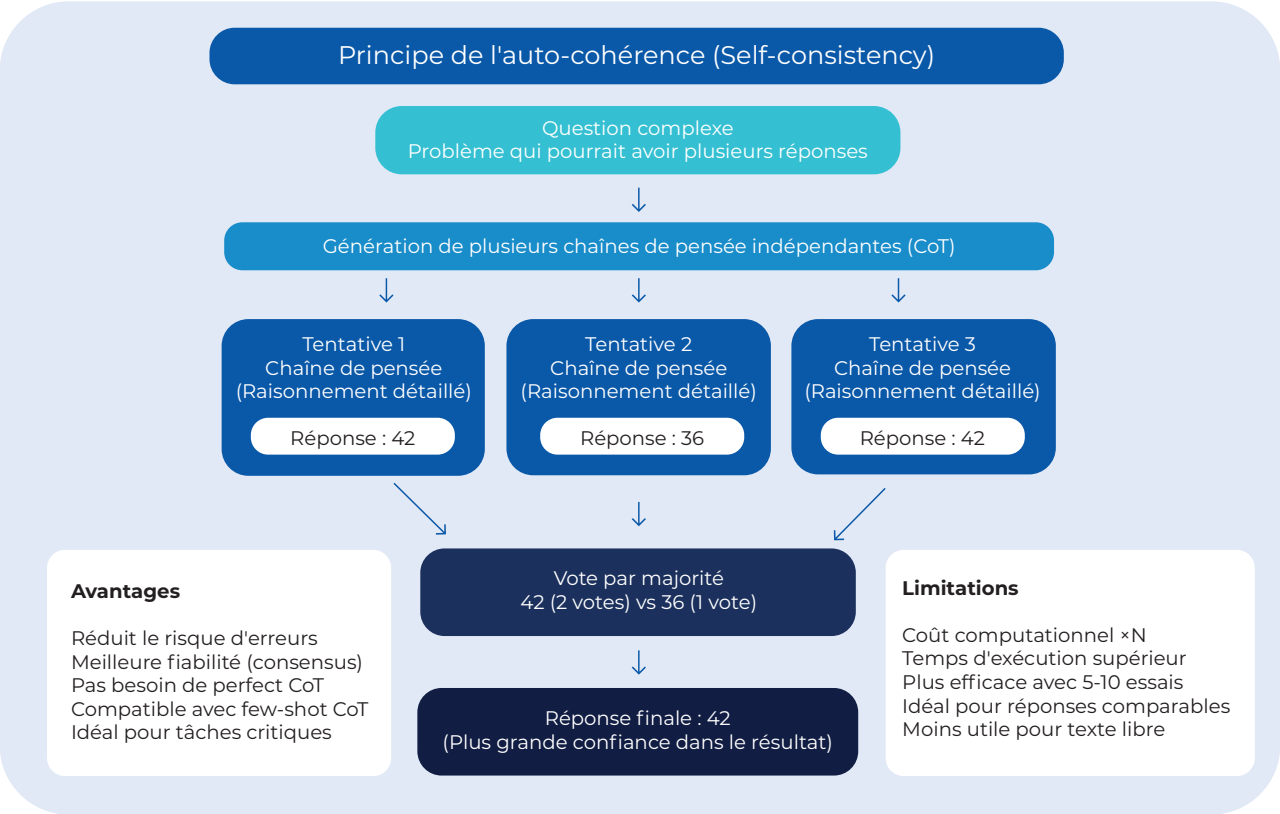
L'intuition sous-jacente est simple : si plusieurs raisonnements variés aboutissent à la **même** conclusion, on peut avoir davantage confiance dans cette conclusion.

Cette approche a démontré des gains de performance significatifs par rapport au *CoT* standard sur de nombreuses tâches de raisonnement. Par exemple, sur un ensemble de problèmes mathématiques scolaires, l'auto-cohérence a pu augmenter le taux de bonnes réponses de près de 18% par rapport à une seule chaîne de pensée. L'auto-cohérence rend aussi le système plus robuste : même si une exécution du modèle tombe sur un raisonnement biaisé ou une hallucination, les autres tentatives peuvent compenser.

### 3.2.2 Coût

Le principal inconvénient est bien sûr le coût computationnel. Si on génère M chemins de pensée, on a besoin d'environ M fois plus de calcul qu'une seule réponse *CoT*. Heureusement, il n'est souvent pas nécessaire d'en générer des dizaines : des études montrent que des votes sur 5 à 10 solutions peuvent déjà apporter un gros bénéfice, le reste ayant des rendements décroissants. Par ailleurs, cette méthode s'applique surtout lorsque la réponse finale est **simple à comparer** d'un essai à l'autre (par exemple un nombre, un choix parmi une liste…). S'il s'agit d'un long texte libre, il n'y a pas de moyen évident de faire un vote de majorité.

**Exemple du principe du vote majoritaire :**




En pratique, l'auto-cohérence est souvent utilisée en coulisse lorsqu'on cherche le **meilleur compromis fiabilité/coût** pour des tâches critiques. Elle offre une assurance supplémentaire que la réponse donnée par le modèle n'est pas un coup de chance isolé, mais qu'elle est cohérente à travers plusieurs trajectoires de raisonnement.

**Astuce :** On peut combiner cette approche avec le few-shot CoT : c'est-à-dire fournir des exemples de chaîne de pensée dans le prompt, et générer plusieurs sorties en échantillonnant. Cela maximise les chances d'obtenir à la fois des raisonnements de qualité et un consensus sur la bonne réponse.
Dans la suite, nous allons découvrir d'autres techniques de raisonnement avancé, notamment comment aborder les problèmes complexes en les résolvant **progressivement** du plus facile au plus difficile, et comment permettre au modèle d'**explorer plusieurs pistes** de solution au lieu d'une seule.

**EXEMPLE**

**Exemple testé d'implémentation pratique :**

« *Je vais te fournir les réponses des différents LLM à une question. Tu devras analyser ces réponses et indiquer celle qui revient le plus souvent. Ta réponse sera uniquement celle-ci, sans aucune information supplémentaire, et devra être formulée ainsi : Ma réponse est...* »

[Réponse 1] :
Pour déterminer combien de « R » contient le mot « strawberry », nous allons suivre ces étapes :
**Écrivons le mot** « strawberry ».
**Comptons chaque occurrence de la lettre « R »** dans le mot.
Voici le mot : strawberry
**Maintenant, comptons les « R » :**
Il y a un « R » dans « strawberry ».
Il y a un autre « R » dans « strawberry ».
Et il y a un troisième « R » dans « strawberry ».
**Donc, le mot « strawberry » contient 3 « R ».**

[Réponse 2] :
Le mot « strawberry » contient 3 « R ».
Voici le raisonnement étape par étape :
**Écrivons le mot** : « strawberry ».
**Comptons le nombre de fois que la lettre « R »** apparaît dans le mot.
**Le mot « strawberry » contient 3 « R ».**

[Réponse 3] :
D'accord ! Raisonnons étape par étape pour déterminer combien de « R » contient le mot « strawberry ».
**Écrivons le mot** : strawberry.
**Comptons chaque lettre « R »** dans le mot.
**Le mot « strawberry » contient 2 « R ».**

[Réponse 4] :
Pour déterminer combien de « R » contient le mot « strawberry », nous allons suivre les étapes :
**Écrivons le mot** « strawberry ».
**Comptons chaque occurrence de la lettre « R »** dans le mot.
Voici le mot : strawberry
**Maintenant, comptons les « R » :**
Il y a un « R » dans « strawberry ».
Il y a un autre « R » dans « strawberry ».
Et il y a un troisième « R » dans « strawberry ».
**Donc, le mot « strawberry » contient 3 « R ».**

[Réponse 5] :
D'accord, raisonnons étape par étape :
Le mot donné est « strawberry ».
**Comptons le nombre de fois que la lettre « R »** apparaît dans ce mot.
**Examinons chaque lettre** du mot « strawberry » : s, t, r, a, w, b, e, r, r, y.
Nous voyons que la lettre « R » apparaît trois fois.
**Donc, le mot « strawberry » contient trois lettres « R ».**

[Réponse finale]
**Ma réponse est : Le mot « strawberry » contient 3 « R ».**

# /Chapitre 4
# Du plus facile au plus difficile – la méthode Least-to-Most

### 4.1 Principes fondamentaux

Dans le chapitre précédent, nous avons vu comment pousser un modèle à dérouler une réflexion étape par étape avec *CoT*. La technique que nous abordons maintenant, appelée ***Least-to-Most (LtM)***, adopte une approche légèrement différente mais complémentaire : elle consiste **à résoudre d'abord des versions simplifiées ou partielles d'un problème, puis à utiliser ces résultats pour traiter progressivement la version complète et complexe du problème.**

En d'autres termes, au lieu de lancer immédiatement le modèle sur le problème le plus difficile, on va le guider à travers une séquence de questions intermédiaires allant du plus **facile** au plus **difficile**, jusqu'à la question finale. Cette méthode s'inspire de stratégies pédagogiques bien connues : pour apprendre une notion complexe, on commence souvent par des cas simples.

### 4.2 Pourquoi utiliser cette méthode

Les *LLM*, comme les humains, peuvent échouer à résoudre directement un problème trop complexe qu'ils n'ont jamais vu, mais ils pourraient y parvenir s'ils ont déjà résolu des sous-problèmes liés. La progression *Least-to-Most* aide le modèle à **généraliser** à des cas plus difficiles en s'appuyant sur des connaissances fraîchement consolidées sur des cas plus simples.

### 4.3 Mise en œuvre

Concrètement, on peut construire un prompt en plusieurs parties, ou enchaîner plusieurs interactions avec le modèle :

- **Étape facile :** poser d'abord une question plus simple en lien avec le problème. Une fois cette partie résolue, récupérer la réponse.
- **Étape intermédiaire :** poser une question un peu plus compliquée, en utilisant la réponse de l'étape précédente comme base ou indice.
- **Étape finale :** poser la question complète/difficile, en s'appuyant sur les éléments obtenus auparavant.

Chaque étape « prépare » le modèle pour la suivante. Notons que cela peut se faire en une seule interaction multi-parties (par ex. fournir un *prompt* où on dit « D'abord fais ceci… puis à partir de ça fais cela… »), ou en plusieurs requêtes successives dans une conversation avec le *LLM*.

### 4.4 Exemples d'application

**Cas d'usage 1 : Identifier la capitale du pays africain le plus peuplé**

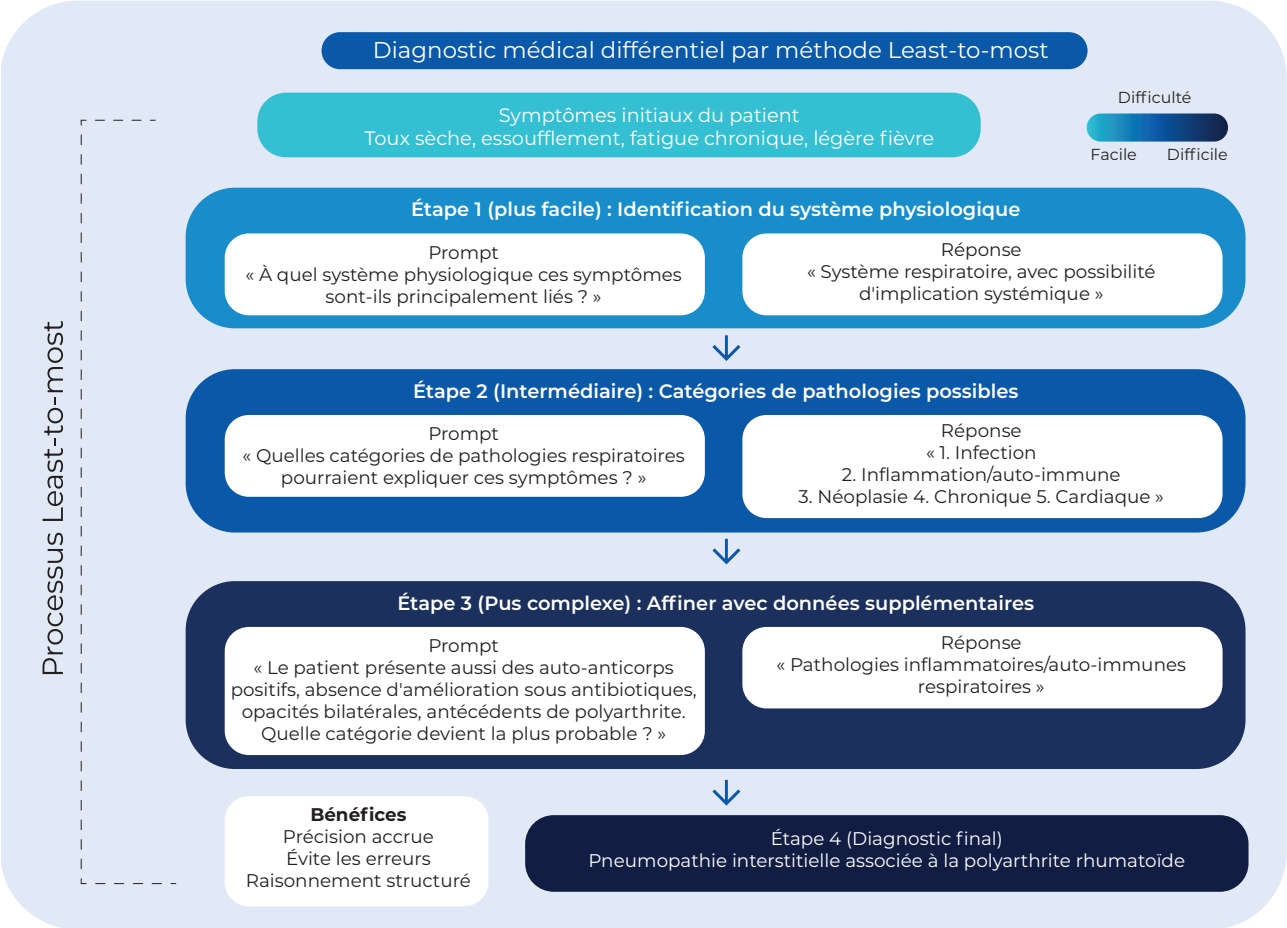
Supposons que notre objectif final est de demander : « *Quelle est la capitale du pays africain le plus peuplé ?* » C'est une question qui combine deux éléments : (a) identifier le pays le plus peuplé d'Afrique, (b) en donner la capitale. On peut aider le modèle à y répondre en deux temps :

**Étape 1 (plus facile) :** « Quel est le pays le plus peuplé d'Afrique ? »
Réponse du modèle : « Le Nigeria est le pays le plus peuplé d'Afrique. »
**Étape 2 (plus facile en utilisant l'étape 1) :** « Quelle est la capitale du Nigeria ? »
Réponse du modèle : « La capitale du Nigeria est Abuja. »

En combinant les deux : on obtient « La capitale du pays africain le plus peuplé (le Nigeria) est Abuja. ».
On aurait pu poser directement la question initiale en une fois. Un modèle très compétent peut répondre du premier coup. Mais un modèle moins sûr aurait pu se tromper (par exemple, confondre avec « Le Caire, Égypte » en pensant que l'Égypte est la plus peuplée). La démarche *LtM* réduit ce risque : à l'étape 1 on s'assure que le modèle a la bonne information (Nigeria), puis l'étape 2 est facile une fois qu'on connaît le pays.

**Cas d'usage 2 : Élaboration d'un diagnostic médical progressif**
Par exemple, face à un tableau clinique complexe avec de multiples symptômes, le modèle peut d'abord identifier le système physiologique concerné, puis déterminer les grandes catégories de pathologies possibles, avant de préciser les diagnostics spécifiques. Cette approche imite le raisonnement médical structuré des cliniciens et réduit significativement les risques d'erreur diagnostique. L'analyse progressive permet au modèle de considérer méthodiquement toutes les informations pertinentes sans se perdre dans la complexité du cas.

**4.5 Avantages**



Des recherches ont montré que la méthode *Least-to-Most* est particulièrement efficace lorsque le modèle doit résoudre des problèmes plus complexes que ceux rencontrés dans les exemples donnés. En effet, dans ces situations, la méthode *Chain of Thought (CoT)* peut échouer, tandis que *Least-to-Most* réussit mieux grâce à sa méthode progressive. En d'autres termes, ***LtM* aide le modèle à gagner en confiance** en commençant par des étapes simples avant de s'attaquer aux problèmes plus difficiles, ce qui lui permet de mieux gérer la complexité croissante et d'appliquer les connaissances acquises à chaque étape précédente pour résoudre les problèmes plus compliqués.

### 4.6 Mise en garde : Les défis de Least-to-Most

La difficulté de *LtM*, c'est qu'il faut savoir quoi demander comme sous-problèmes. Cela requiert de notre part une compréhension du problème initial et une capacité à le décomposer intelligemment. Ce n'est pas toujours évident de déterminer les bonnes étapes intermédiaires. Cependant, avec l'expérience, on apprend à repérer les sous-tâches naturelles d'un problème et à formuler ainsi des *prompts* séquentiels efficaces.

En résumé, *Least-to-Most* est une technique de *prompt engineering* qui imite une progression pédagogique. Elle s'avère utile pour guider un *LLM* à travers un raisonnement ou une tâche en plusieurs paliers de difficulté croissante. Maintenant que nous avons vu les approches linéaires *(CoT et LtM)* pour faire raisonner le modèle, intéressons-nous à une idée encore plus ambitieuse : permettre au modèle d'explorer **plusieurs pistes de raisonnement en parallèle**, grâce à l'Arbre de Pensées.

# /Chapitre 5
# Explorer plusieurs pistes avec l'Arbre de Pensées (Tree-of-Thoughts)

### 5.1 Introduction au concept

Jusqu'ici, nos techniques *(CoT, LtM)* faisaient suivre au modèle un **chemin linéaire** de réflexion. Mais face à certaines énigmes ou tâches complexes, on gagnerait à pouvoir essayer **plusieurs approches alternatives** et voir laquelle fonctionne le mieux. C'est exactement le but de la méthode de l'**Arbre de Pensées** *(Tree-of-Thoughts, ToT)* .

Dans un *Tree-of-Thoughts*, on ne se limite plus à une seule séquence d'étapes. À certaines étapes clés, le modèle est autorisé à **bifurquer** et explorer différentes options ou idées, créant ainsi une sorte d'arborescence de réflexion. Ensuite, grâce à une stratégie d'évaluation, on décide quelles branches de l'arbre valent la peine d'être approfondies. En résumé, *ToT* généralise *CoT* en un processus de recherche arborescente plutôt que linéaire.

### 5.2 Cas d'utilisation

Imaginez un casse-tête logique ou une énigme où il faut éventuellement faire des essais-erreurs. Avec *CoT* seul, si le modèle part sur la mauvaise piste dès le début, il ira au bout de cette piste et donnera une réponse qui pourrait être fausse, sans jamais revenir en arrière. Avec *ToT*, le modèle peut se dire : « *À ce point du raisonnement, j'entrevois deux possibilités plausibles, explorons-les tour à tour* ». C'est particulièrement indiqué pour des problèmes de **planification, de jeu, ou de recherche de solution** où il faut examiner plusieurs possibilités. Par exemple, résoudre un puzzle qui nécessite de tester différentes combinaisons, ou planifier des étapes où chacune peut être faite de différentes manières.

### 5.3 Fonctionnement et mise en œuvre

On peut implémenter un *Tree-of-Thoughts* de différentes façons, mais voici un schéma conceptuel :

**1 -** Le modèle commence à réfléchir et génère une première étape.

**2 -** À l'étape suivante, au lieu de produire une seule continuation, il génère **plusieurs suites possibles** (plusieurs idées ou actions possibles). Ces différentes continuations forment des branches.

**3 -** Pour chacune de ces branches, il continue le raisonnement (profondeur suivante de l'arbre), potentiellement bifurque à nouveau plus loin, etc.

**4 -** À un certain point, on doit **évaluer les branches** – soit via une fonction d'évaluation externe (par ex. tester si une solution atteint l'objectif), soit en demandant au modèle d'évaluer laquelle de ses branches semble la plus prometteuse.

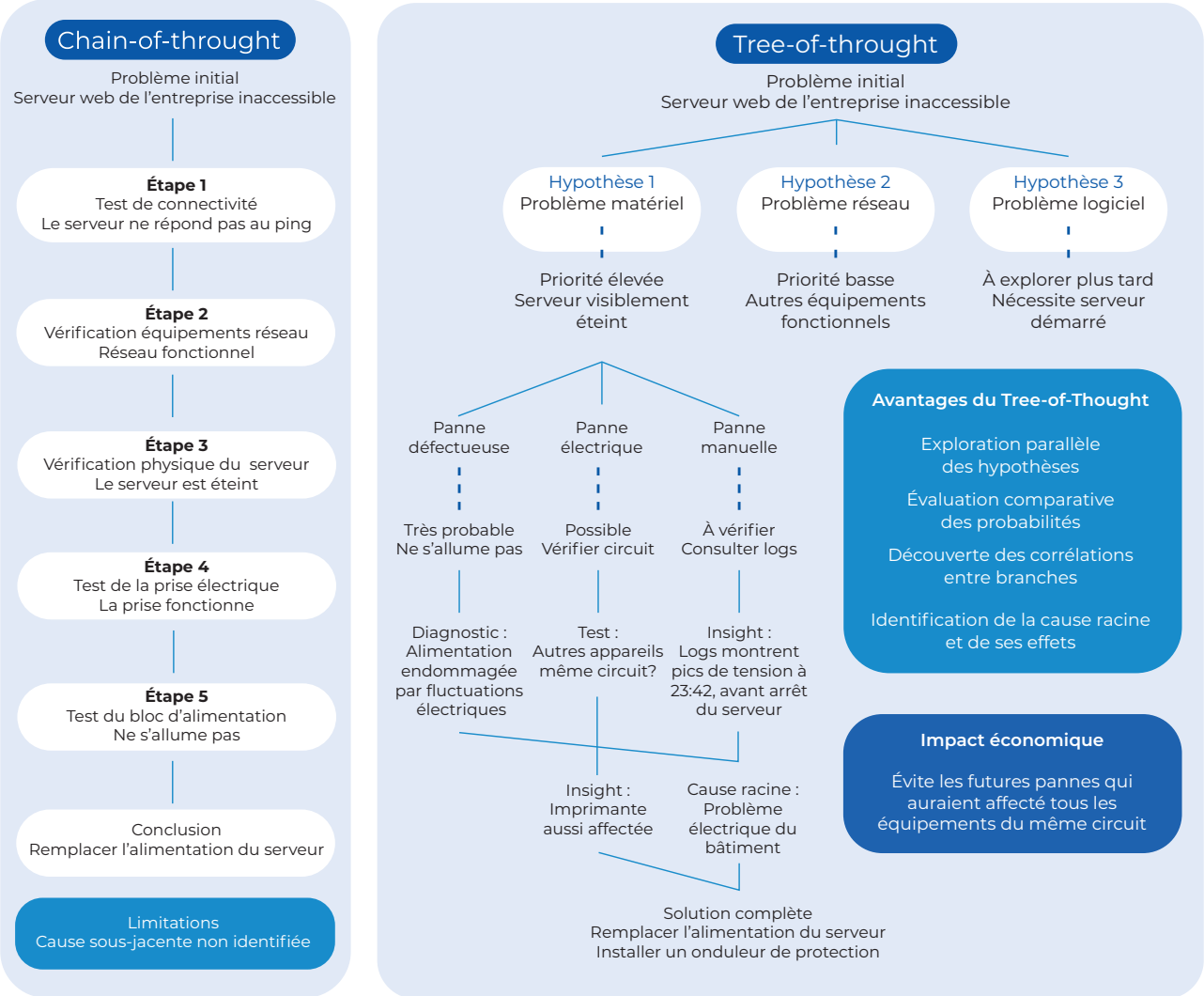
**5 -** On peut alors **élaguer** les branches jugées non pertinentes et continuer à développer les branches prometteuses. Ce processus peut continuer sur plusieurs niveaux jusqu'à trouver une solution satisfaisante ou jusqu'à épuiser une profondeur fixée.

### 5.4 Orchestration et implémentation

Vu de l'extérieur, orchestrer un *Tree-of-Thoughts* peut ressembler à faire interagir le *LLM* avec un algorithme de recherche. On peut, par exemple, piloter la génération en boucle : demander « Donne-moi 3 idées pour la prochaine étape », puis « Évalue ces 3 idées », puis « Développe l'idée 2 plus en détail » et ainsi de suite. Cela requiert un *prompt engineering* plus complexe, souvent multi-tour, et potentiellement l'aide d'une fonction de *scoring*.

**Un avantage mesuré** de *ToT*, c'est que sur certains problèmes de type jeu ou casse-tête (par ex. résoudre le jeu du 24 où il faut trouver comment obtenir 24 à partir de certains chiffres), *ToT* surpasse largement *CoT*. En effet, *ToT* permet d'essayer plusieurs combinaisons d'opérations, alors qu'un CoT, linéaire, risque de s'enliser s'il ne prend pas la bonne combinaison du premier coup.

### 5.5 Analogies explicatives



**Comparaison stratégique**

On peut comparer *CoT vs ToT* à la différence entre un joueur d'échecs débutant qui envisage un seul enchaînement de coups possible (et le suit tête baissée), et un joueur avancé qui examine plusieurs coups possibles à chaque tour et anticipe plusieurs scénarios. Le second a bien sûr plus de chances de trouver une stratégie gagnante.

**Métaphore du labyrinthe**

Imaginez un modèle qui doit résoudre un labyrinthe. Avec une approche *CoT* simple, il pourrait choisir une direction et avancer tout droit ; s'il se trompe, tant pis, il finira dans un cul-de-sac sans solution. Avec *ToT*, c'est comme si le modèle pouvait dire : « à cette intersection, je peux aller à gauche ou à droite ; je vais explorer la voie de gauche, voir où elle mène… et aussi garder en tête la voie de droite pour l'essayer si la gauche échoue. » On permet donc au modèle de **revenir en arrière** sur le plan de la réflexion, ce que *CoT* ne faisait pas.

**Limites et complexité**

La mise en place d'un **Tree-of-Thoughts** est plus complexe. Il faut gérer potentiellement un grand arbre (le nombre de branches peut exploser si on n'y prend garde), et il faut une méthode pour évaluer les branches (ce qui peut parfois être fait par le *LLM* lui-même en le faisant noter ses idées, ou par un critère programmé). *ToT* peut être vu comme un cadre général de recherche, et il peut être assisté par des techniques algorithmiques classiques (par ex. des algorithmes de parcours en largeur/profondeur couplés au *LLM*). C'est une approche de pointe, plus lente et coûteuse que *CoT*, à réserver pour des problèmes vraiment ardus qui justifient ces efforts.

## 5.6 Extension : le Graphe de Pensées (Graph-of-Thoughts)

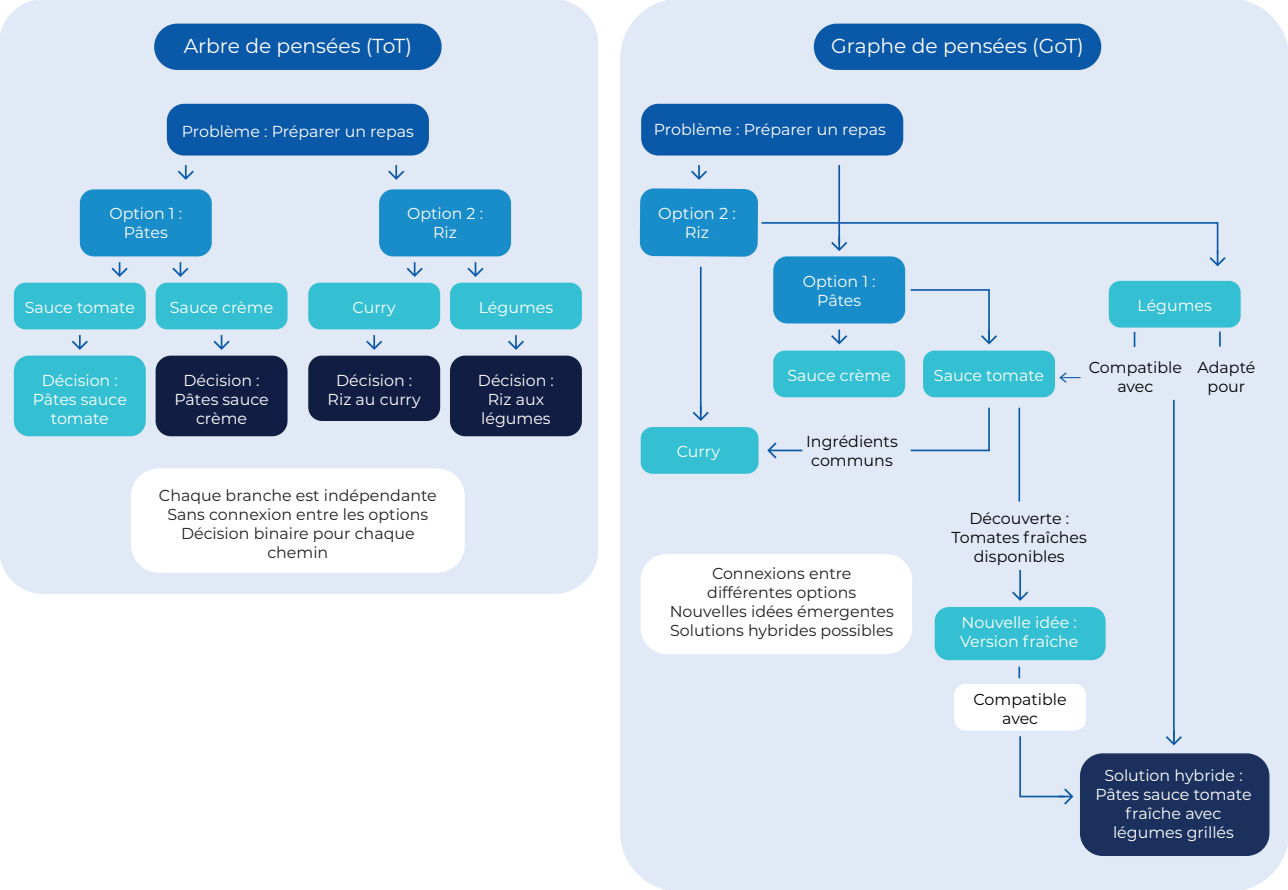
**Concept et avantages**

Que pourrait-il y avoir de plus général qu'un arbre ? Un graphe arbitraire ! En poussant l'idée encore plus loin, des chercheurs ont proposé le concept de **Graphe de Pensées** *(Graph-of-Thoughts, GoT)*. Dans un graphe, les chemins de raisonnement ne forment pas nécessairement une hiérarchie arborescente ; ils peuvent se rejoindre, boucler, diverger de manière plus souple qu'un arbre.

Cela pourrait permettre, par exemple, de fusionner deux branches de pensée qui se révèlent complémentaires, ou de revenir à un état antérieur du raisonnement pour emprunter une autre direction (ce qu'un arbre strict ne fait pas une fois une branche abandonnée).

**Arbre vs graphe de pensées**

Exemple simple : Préparer un repas




**État actuel de la recherche**

Le Graphe de Pensées est pour l'instant un concept exploratoire et avancé. On peut l'envisager comme un **filet de sécurité ultime** pour le raisonnement du *LLM* : il modéliserait de manière très générale la façon dont on pourrait naviguer à travers différentes idées, avec la possibilité de revenir en arrière et de réutiliser des morceaux de raisonnement d'une branche dans une autre.

**Implications pratiques**

Pour un *prompt engineer*, il n'y a pas encore de recette pratique standard pour « faire du Graphe de Pensées ». C'est plus une direction de recherche qu'une technique *plug-and-play.* Retenez simplement que la communauté cherche à donner aux *LLM* des capacités de **raisonnement toujours plus flexibles**, et qu'après l'arbre, le graphe est la prochaine métaphore naturelle.

**5.7 Résumé et perspectives**

*Tree-of-Thoughts (ToT)* introduit la notion d'**exploration stratégique** de plusieurs voies de raisonnement. Là où *CoT* suit un chemin unique, *ToT* permet de bifurquer et de tester différentes approches – un peu comme un cerveau collectif où le modèle pourrait essayer plusieurs solutions en parallèle. C'est une technique puissante pour les problèmes nécessitant planification, essai-erreur ou imagination de scénarios multiples. Son extension, *Graph-of-Thoughts*, reste expérimentale mais ouvre la voie à des raisonnements encore plus

souples.

À présent, nous allons sortir du cadre du raisonnement interne du modèle pour explorer des techniques où le modèle peut i**nteragir avec des sources externes.** En effet, un *LLM* peut très bien raisonner, mais ne peut pas inventer une information qu'il n'a pas. Dans le prochain chapitre, nous verrons comment des approches permettent à l'IA d'aller chercher de l'information ou d'agir dans un environnement, grâce à des paradigmes comme ***ReAct* et *RAG*.**

# /Chapitre 6
# Interagir avec l'extérieur – le framework ReAct et la génération augmentée par des ressources (RAG)

Jusqu'à présent, toutes nos techniques (du *prompt* de base jusqu'à l'arbre de pensées) se déroulaient **entièrement dans la tête du modèle.** Or, les **LLM** – même très grands – opèrent en vase clos : leurs connaissances proviennent de ce qu'ils ont vu en entraînement (ce qui peut être obsolète ou incomplet), et ils n'interagissent pas naturellement avec des outils ou des bases de données externes. Que faire si la question posée dépasse les connaissances mémorisées du modèle ? Ou si résoudre une tâche nécessite d'effectuer une action dans le monde réel (consulter un site web, exécuter un calcul, etc.) ?

Pour répondre à ce besoin, on a développé des techniques de ***prompting* interactif,** où le modèle est guidé pour effectuer des actions externes en plus de la génération de texte. Les deux approches phares dans ce domaine sont ***ReAct*** *(Reason + Act)* et ***RAG*** *(Retrieval-Augmented Generation).*

## 6.1 ReAct : raisonner et agir de concert (Reason + Act)

### 6.1.1 Définition et principes fondamentaux

Le *framework ReAct* est conçu pour intégrer de manière synergique le raisonnement interne et les actions externes d'un *LLM*. Au lieu de produire uniquement des « pensées », (du texte intermédiaire) ou uniquement des actions, le modèle fait les deux en alternance. En pratique, on le pousse à générer des cycles du type : Pensée → Action → Observation → ... → (répéter), jusqu'à ce qu'une réponse finale soit prête.

Ce cycle se compose de trois éléments essentiels qui fonctionnent en synergie :

- **Pensée *(Thought)* :** le *LLM* génère une réflexion sur l'état actuel du problème. Cela peut être une analyse de ce qu'il sait, une hypothèse, ou la détection d'un manque d'information. Par exemple : « *Hmm, la question porte sur la population de Paris en 2025, je ne suis pas sûr de la connaître. Je devrais peut-être chercher cette information.* »
- **Action *(Act)* :** le *LLM* produit une instruction d'action. Cela peut être formulé dans le texte de sortie d'une manière conventionnelle, par exemple : « Action: *Search*[Population Paris 2025] », – qui est ensuite interceptée par le système extérieur. L'action pourrait être de faire une recherche web, de consulter une base de connaissances, d'appeler une API de calcul, etc.
- **Observation *(Observation)* :** le résultat de l'action est renvoyé au modèle. Par exemple, après la recherche on fournit au modèle un extrait d'article disant « *En 2025, la population de Paris est estimée à X habitants.* »

Le cycle peut alors reprendre : fort de cette observation, le modèle peut formuler une nouvelle pensée (« *D'accord, j'ai trouvé la population, je vais maintenant répondre à la question.* ») ou décider d'une autre

action si nécessaire. Ce paradigme pensée-action se répète jusqu'à ce que le modèle ait suffisamment d'informations pour donner la réponse finale qu'on attend de lui.

### 6.1.2 Exemple concret d'utilisation

Supposons qu'on pose au modèle : « *Quelle est la population de Paris en 2023 ?* » (et imaginons que son entraînement date de 2021, il ne sait donc pas cette information).

**Un déroulement ReAct pourrait être :**

- **Pensée :** « La question porte sur la population de Paris en 2023. Je n'ai pas cette donnée précise. Je vais chercher sur internet »
- **Action :** (Le modèle émet une commande) [Recherche : « population Paris 2023 »]
- **Observation :** (Le système fournit le résultat de la recherche) – disons un extrait d'article : « ...En 2023, la population de la ville de Paris est d'environ 2,165 millions d'habitants... ».
- **Pensée :** « Je viens de trouver que Paris a ~2,165 millions d'habitants en 2023. C'est l'information qu'il me fallait pour répondre précisément. »
- **Réponse finale** : « En 2023, Paris compte environ 2,165 000 habitants. »

Dans cet exemple, sans *ReAct*, le modèle aurait probablement tenté de répondre avec des connaissances dépassées (par ex. la population en 2019) ou une estimation floue. Grâce à la recherche, il fournit la bonne réponse avec la donnée à jour. On voit l'alternance pensée/action : la Pensée justifie l'Action, l'Observation alimente la pensée suivante, etc.

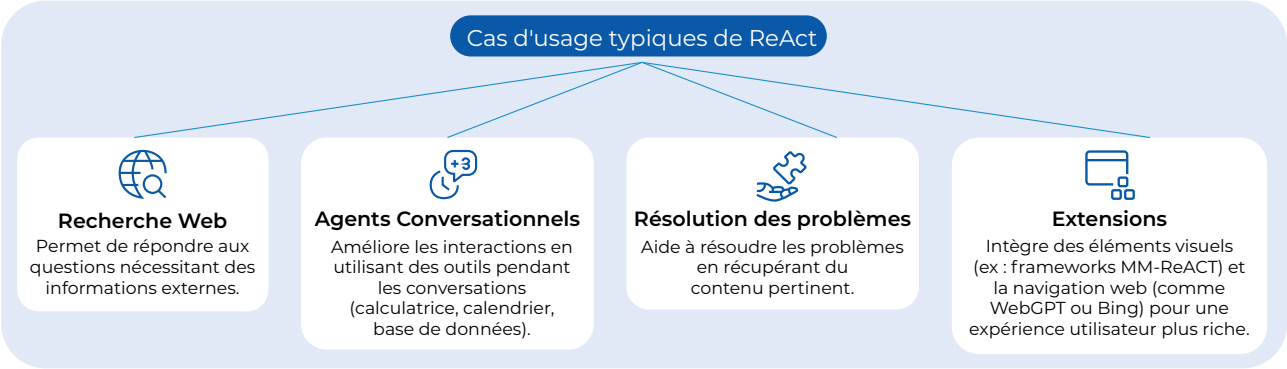
### 6.1.3 Avantages et cas d'usage

La puissance de *ReAct* réside dans sa capacité à transformer le *LLM* d'un simple générateur de texte en un agent capable d'interagir avec son environnement. *ReAct* excelle pour les tâches nécessitant une recherche d'information dynamique ou une manipulation d'environnement.
Par exemple, pour répondre à une question d'actualité, le modèle seul hallucinerait sans doute une réponse ; en mode *ReAct*, il va chercher la réponse et ne la fournir qu'après l'avoir trouvée, réduisant drastiquement les risques d'erreurs factuelles. De même, pour résoudre un problème de calcul complexe, le modèle peut décider d'appeler une calculatrice externe plutôt que d'essayer de tout calculer lui-même.

Des travaux ont montré que *ReAct* surpasse les approches qui soit raisonnaient sans agir, soit agissaient sans raisonner : c'est la combinaison qui fait la force du *framework*. En effet, un agent *ReAct* ne se contente pas d'agir au hasard : il planifie ses actions via ses pensées, ce qui les rend bien plus efficaces.

Pour activer le **framework ReAct**, un exemple de prompt pourrait être formulé comme suit :

**Prompt exemple :** « *Réfléchissons étape par étape pour répondre à la question suivante : Quelle est la population de Paris en 2023 ? Si tu ne connais pas la réponse, commence par rechercher l'information sur internet et fais-moi savoir ce que tu trouves. Je vais m'assurer que tu aies suffisamment d'informations pour répondre correctement.* »




## 6.2 RAG : Récupérer des ressources et enrichir les réponses (Retrieval Augmented Generation)

### Concept et principes fondamentaux

La deuxième grande approche pour combler les lacunes des modèles est la **génération augmentée par récupération d'informations**, en abrégé *RAG*. L'idée est un peu plus simple que *ReAct* : avant ou pendant que le *LLM* génère sa réponse, on lui fournit activement des **données externes pertinentes** issues d'une base de connaissances ou d'un moteur de recherche. Le *LLM* utilise alors cette **connaissance fraîche** pour produire sa réponse, au lieu de se reposer uniquement sur sa mémoire interne.

### Architecture en deux étapes

En pratique, cela se traduit souvent par une architecture en deux étapes :

**1 - *Retrieval (Récupération)* :** À partir de la question de l'utilisateur, on effectue une recherche dans une grande base de textes (par exemple Wikipédia, des documents d'entreprise, ou même le web). On sélectionne les passages les plus pertinents (par une méthode de similarité sémantique, de recherche plein texte, etc.).

**2 - *Generation (Génération)* :** On construit un prompt qui contient la question de l'utilisateur ainsi que les passages de texte récupérés comme contexte, puis on demande au *LLM* de répondre en s'appuyant sur ces documents. Le modèle va ainsi traiter ces informations comme faisant partie de son contexte immédiat, ce qui lui permet de fournir une réponse à jour et ancrée sur ces données externes.

### Avantages et bénéfices

On parle d'« augmentation » car le modèle voit son *prompt* augmenté de connaissances supplémentaires qu'il n'avait pas stockées en lui-même. Ce procédé est utilisé par exemple par les systèmes comme les versions avancées de ChatGPT/Bing lorsqu'ils citent des sources. En coulisse, ils ont intégré dans le *prompt* des extraits d'articles relatifs à la question posée, puis le modèle rédige une réponse en utilisant ces extraits.

**EXEMPLE** **Exemple d'application**

**Question utilisateur :** « *Quel était le discours principal prononcé par Martin Luther King Jr. en 1963 et quel en est le thème principal ?* »

- **Phase de récupération :** Le système effectue une recherche et trouve un document (ou passage) mentionnant « Martin Luther King Jr., discours *'I Have a Dream'*, 1963, thème : l'égalité raciale et la liberté. »
- **Phase de génération :** Le prompt fourni au modèle pourrait être construit ainsi : « [*DOCUMENT]: En août 1963, MLK a prononcé le célèbre discours 'I Have a Dream' lors de la Marche sur Washington. Ce discours portait principalement sur la nécessité de l'égalité raciale et la fin des discriminations... [QUESTION UTILISATEUR]: Quel était le discours principal prononcé par MLK en 1963 et quel en est le thème principal ?* »

Avec ces informations dans son contexte, le modèle répondra : « Le discours principal de Martin Luther King Jr. en 1963 est ***« I Have a Dream »***, et son thème central est l'égalité raciale et la liberté, appelant à la fin du racisme et des discriminations aux États-Unis. ».

Sans *RAG*, un modèle pourrait se souvenir du titre du discours, mais il pourrait mal formuler le thème, ou manquer de détails. Avec *RAG*, la réponse est calée sur le contenu du document récupéré.

**Mise en œuvre pratique**

RAG peut être réalisé de diverses façons. Une approche consiste à créer un **vecteur sémantique** de la question puis chercher les vecteurs proches dans une base de données de connaissances (c'est la méthode des *embeddings cosinus*, etc.), ou simplement à utiliser un moteur de recherche classique en construisant une requête textuelle. Une fois les extraits obtenus, on les incorpore dans le *prompt* (en général sous forme de citations ou d'annexes). Il faut faire attention à la longueur (si trop de texte est injecté, on peut saturer la fenêtre de contexte). Parfois, on utilise aussi le modèle pour générer une reformulation de la question qui améliore la recherche (on « *prompt* » un moteur de recherche via le *LLM*).

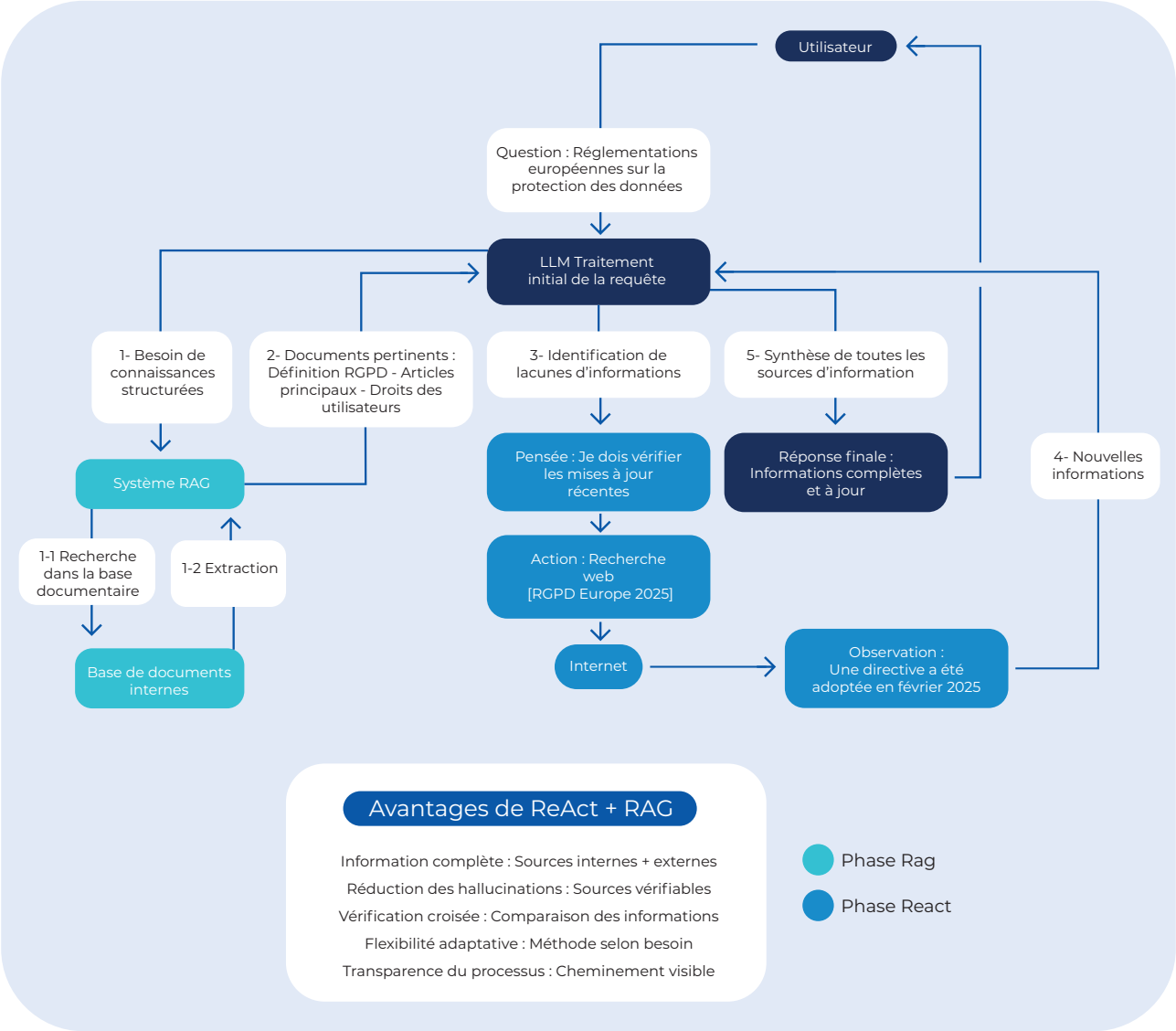
**Comparaison ReAct vs RAG**

ReAct et RAG ont un objectif commun (combler les limites internes du *LLM* en l'ouvrant sur l'extérieur) mais des approches un peu différentes. ReAct fait dialoguer le *LLM* avec des outils étape par étape, alors que RAG fait un pré-travail de récupération d'infos qu'on donne en masse au *LLM* avant génération. On pourrait dire que ReAct est plus **procédural** (permet plusieurs allers-retours actions/réflexion), tandis que RAG est plus **informatif immédiat** (on fournit les connaissances une bonne fois et on rédige).

**Combinaison des approches**

Ces deux approches peuvent d'ailleurs être combinées : par exemple un agent ReAct peut utiliser un outil de recherche pour trouver des documents, puis les insérer dans son prompt (c'est une forme de RAG initiée via ReAct). L'important pour nous est de comprendre que le prompt engineering ne se limite pas au texte statique qu'on donne au modèle initialement : on peut concevoir des systèmes où le prompt évolue

dynamiquement en intégrant des **observations** (résultats de recherches, etc.) sur lesquelles le modèle va s'appuyer.




**En résumé**, ReAct et RAG illustrent une évolution significative dans l'utilisation des *LLM* : on reconnaît que les connaissances internes du modèle, bien qu'immenses, sont statiques et parfois incomplètes. Ces techniques utilisent l'interaction avec l'extérieur – actions ciblées pour ReAct, ou récupération documentaire pour RAG – pour pallier ces limitations. ReAct permet au modèle de **faire des actions** (comme chercher, calculer, interagir avec un environnement) afin de tester et affiner son raisonnement en temps réel. RAG, quant à lui, fournit au modèle un **complément de connaissances** sur lequel il peut s'appuyer lors de la génération. Grâce à elles, un *LLM* n'est plus un simple générateur isolé, mais peut devenir un agent connecté à des informations et capable d'**augmenter** ses réponses par des données à jour et vérifiées.

Après avoir vu comment un modèle peut agir et s'informer, intéressons-nous à d'autres techniques avancées qui se concentrent sur l'**amélioration du processus de génération lui-même.** Nous allons découvrir notamment le concept de scratchpad, qui aide le modèle à tenir un carnet de notes interne, et des méthodes pour que le modèle se corrige ou s'optimise lui-même.

# /Chapitre 7
# Garder une trace – Scratchpad et Chaîne de Brouillons (CoD)

Lorsque les *LLM* doivent effectuer des tâches complexes comme des calculs, du code ou des raisonnements très détaillés, ils peuvent avoir du mal à garder en tête toutes les informations intermédiaires. Tout comme nous utilisons un brouillon papier pour poser nos calculs ou noter des éléments pendant un problème, on peut encourager un modèle à **écrire ses étapes intermédiaires quelque part** avant de donner sa réponse finale. C'est l'idée du *Scratchpad Prompting*, ou *prompt* à brouillon.

## 7.1 Scratchpad : le bloc-notes du modèle

Un *scratchpad* est, en quelque sorte, une section du *prompt* ou de la réponse où le modèle a le droit de déposer des calculs, des variables, des pensées structurées qui ne seront pas directement interprétées comme la réponse finale, mais comme du **travail intermédiaire**. C'est très proche conceptuellement de *Chain-of-Thought*, à la différence près que le *scratchpad* peut être plus formel ou structuré (par exemple, il peut prendre la forme d'un tableau de valeurs intermédiaires, ou de pseudo-code en cours d'exécution, etc.).

### 7.1.1 Comment l'utiliser ?

Souvent, on incite le modèle à utiliser un *scratchpad* en le spécifiant dans le *prompt*. Par exemple : « Montre tes calculs pas à pas, puis donne la réponse finale à la fin. » ou « Utilise le format suivant : d'abord une section « Calcul intermédiaire », puis la réponse. ». On peut même donner un exemple formaté d'utilisation du *scratchpad*.

Le *scratchpad* est particulièrement utile pour les tâches de **raisonnement algorithmique** ou de **calcul** où il y a un état qui évolue étape par étape. Des recherches ont montré qu'en encourageant un *LLM* à externaliser ses états intermédiaires (par exemple, afficher les variables modifiées à chaque étape d'un code), on améliorait notablement sa capacité à suivre des procédures longues sans erreur.

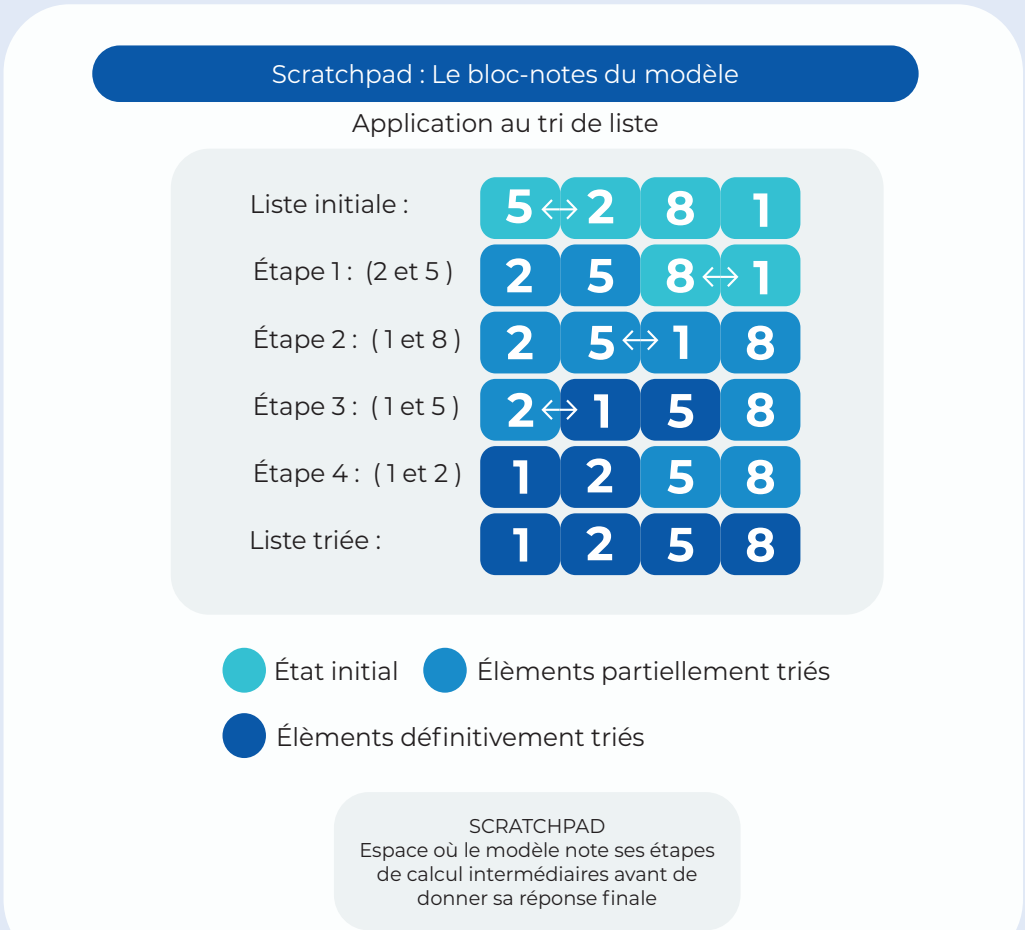
### 7.1.2 Exemples

**EXEMPLE 1**

Si on demande au modèle de faire un tri dans une liste en montrant les étapes, un *scratchpad* approprié pourrait être quelque chose comme :

**Prompt :** *Trie la liste suivante par ordre croissant, affiche dans ta réponse le contenu de ton scratchpad [5, 2, 8, 1]*

**Réponse :** Voici le tri de la liste [5, 2, 8, 1] par ordre croissant, avec le contenu du scratchpad :




Ici, le *scratchpad* est la série d'états de la liste à chaque étape de l'algorithme de tri, avant de donner la liste triée finale.

**EXEMPLE 2**

**Prompt :** *Quels sont les trois principaux avantages d'utiliser l'énergie solaire pour un particulier ?*

**Réponse avec ScratchPad implicite :**

**1 -** Réduction des factures d'électricité.

**2 -** Contribution à la protection de l'environnement.

**3 -** Indépendance énergétique accrue.

**Explication de l'utilisation du ScratchPad (interne à l'IA) :**

Pour arriver à cette réponse concise, une IA utilisant un ScratchPad pourrait avoir suivi un processus de raisonnement interne comme celui-ci :

**Requête interne (simulée) :** « Identifier les avantages de l'énergie solaire pour un particulier. »

**Exploration de connaissances (dans sa base de données) :**

- **Information 1 :** L'énergie solaire réduit la dépendance au réseau électrique et donc les coûts associés. (Note : Potentiel avantage - Réduction des coûts)
- **Information 2 :** L'énergie solaire est une source d'énergie renouvelable qui ne produit pas de gaz à effet de serre pendant son fonctionnement. (Note : Potentiel avantage - Impact environnemental positif)
- **Information 3 :** L'installation de panneaux solaires peut rendre un foyer moins dépendant des fluctuations des prix de l'énergie. (Note : Potentiel avantage - Indépendance énergétique)
- **Information 4 :** Il existe parfois des incitations financières ou des subventions pour l'installation de systèmes solaires. (Note : Potentiel avantage - Économies initiales - pourrait être combiné avec la réduction des factures)
- **Information 5 :** La valeur d'une propriété équipée de panneaux solaires peut augmenter. (Note : Potentiel avantage - Augmentation de la valeur immobilière - moins direct comme avantage principal)

**Tri et Priorisation (dans le ScratchPad) :** L'IA évalue les avantages potentiels en fonction de leur importance et de leur pertinence directe pour un particulier. Elle pourrait décider de se concentrer sur les aspects financiers récurrents, l'impact environnemental direct et l'autonomie.

**Synthèse et Formulation de la Réponse (en utilisant les notes du ScratchPad) :** L'IA formule les trois principaux avantages de manière concise et claire.

Dans cet exemple, le « *ScratchPad* » a servi à :

- Stocker temporairement des informations extraites de sa connaissance.
- Évaluer et prioriser ces informations en fonction de la question.
- Synthétiser les points clés pour former la réponse finale.

L'utilisateur ne voit que le résultat final, mais en interne, l'IA a effectué un travail de traitement et de sélection d'informations qui peut être assimilé à l'utilisation d'un brouillon. L'accent est mis sur la fourniture des avantages les plus pertinents sans détailler tout le processus de recherche et de tri interne.

### 7.1.3 Avantages et limites

Le *scratchpad* aide le modèle à structurer sa pensée et à ne pas perdre de vue des détails. C'est aussi utile pour nous, car on peut suivre ce qu'il fait. On peut presque le guider à travers le format de *scratchpad*. Par exemple, si on attend certaines étapes précises, on peut les mentionner (comme dans l'exemple du tri). Une étude a montré que le *scratchpad*, combiné à un entraînement approprié, permettait à des modèles de résoudre des problèmes de code plus longs qu'ils n'auraient pu sinon.

Tous les modèles ne savent pas spontanément utiliser un *scratchpad* correctement. Souvent, la technique est la plus efficace quand le modèle a été *fine-tuné* pour (c'est-à-dire entraîné avec des exemples où il y avait un *scratchpad*). Si ce n'est pas le cas, on peut tout de même essayer de le forcer via le *prompt*, mais le résultat peut varier. De plus, un *scratchpad* très verbeux consomme des tokens, il y a donc un coût en longueur.

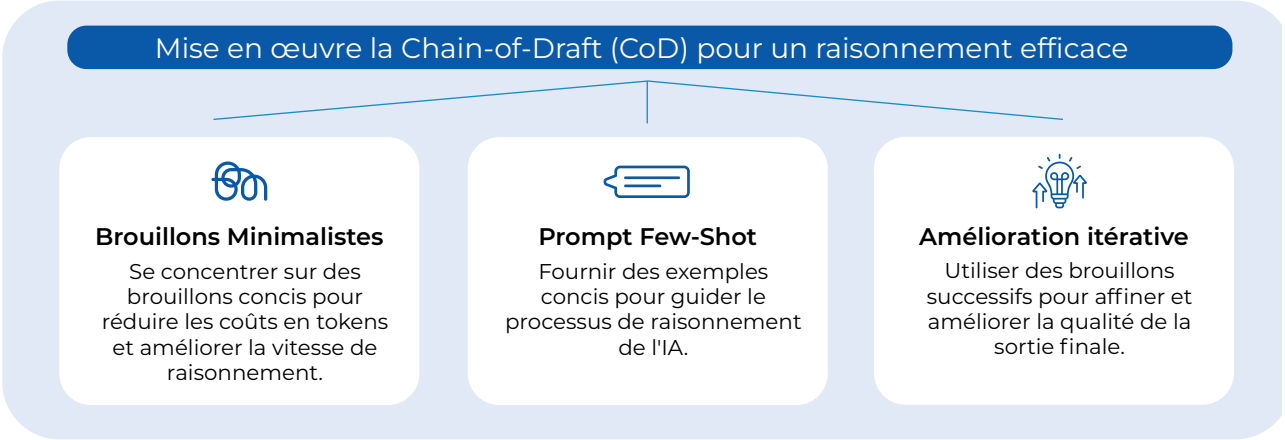
## 7.2 Chain-of-Draft (CoD) : penser plus vite en écrivant moins

### 7.2.1 Principes fondamentaux de la Chain-of-Draft (CoD)

La *Chain-of-Draft (CoD)*, ou « Chaîne de Brouillons », est une évolution récente du scratchpad/CoT, visant à rendre le processus de raisonnement moins coûteux en *tokens*. Plutôt que de rédiger de longues phrases détaillées, le modèle produit des brouillons minimalistes à chaque étape, ne contenant que l'information cruciale nécessaire pour avancer vers la solution.

Ce processus se matérialise par exemple par l'écriture d'un résultat numérique intermédiaire ou la mise à jour d'une variable, suivis immédiatement de l'étape suivante. On peut imposer des contraintes telles que « *Ne dépasse pas 5 mots par étape de raisonnement. Contente-toi de l'essentiel.* »

**Concept Clé :** La Chaîne de Brouillon simule un processus d'écriture itératif où l'on ne vise pas la perfection dès le premier jet, mais plutôt une amélioration continue à travers plusieurs ébauches. L'IA génère un premier brouillon, puis le révise, l'étend, le corrige et le modifie pour produire une version finale plus aboutie.
*CoD* se réalise souvent via du *few-shot prompt* également : on montre dans le *prompt* des exemples où les raisonnements intermédiaires sont très concis.



### 7.2.2 Exemple de mise en œuvre : comparaison CoT vs CoD

**Scénario :** Un utilisateur demande à l'IA de l'aider à planifier un itinéraire de voyage de 3 jours à Rome, en incluant des sites historiques majeurs, des options de restauration typiques et des conseils pour éviter les foules.

**Réponse utilisant la Chaîne de Pensée :**

**1 -** Décomposer le problème : Planifier 3 jours, inclure sites historiques, restauration typique, conseils anti-foules.

**2 -** Identifier les sites historiques majeurs (connaissance) : Colisée, Forum Romain, Vatican (Basilique Saint-Pierre, Musées du Vatican, Chapelle Sixtine), Panthéon, Fontaine de Trevi, Place d'Espagne.

**3 -** Distribuer les sites sur 3 jours (logique et contraintes de temps/proximité) :

- **Jour 1 :** Colisée et Forum Romain (proximité), Panthéon, Fontaine de Trevi, Place d'Espagne (centre historique).
- **Jour 2 :** Vatican (journée entière).
- **Jour 3 :** Sites potentiels restants ou activités alternatives (Borghese Gallery si réservation, Trastevere).

**4 - Suggérer des options de restauration typiques (connaissance) :** Pizzerias, trattorias pour pâtes (cacio e pepe, carbonara, amatriciana), gelaterias. Mentionner des quartiers comme Trastevere pour une ambiance authentique.

**5 - Fournir des conseils anti-foules (connaissance et logique) :** Visiter tôt le matin ou tard l'après-midi, éviter les week-ends et les périodes de vacances, réserver les billets à l'avance (Vatican, Colisée), explorer les quartiers moins touristiques.

**6 - Formuler l'itinéraire et les conseils de manière structurée (texte) :** Présenter un plan jour par jour avec les sites, des suggestions de restaurants et une section de conseils.

**Explication de la Chaîne de Pensée :**

L'IA a décomposé la requête complexe en sous-problèmes gérables. Elle a utilisé sa connaissance des sites touristiques, de la gastronomie locale et des stratégies pour éviter les foules. Ensuite, elle a appliqué une logique de planification (proximité géographique, temps nécessaire pour chaque visite) pour structurer l'itinéraire et a formulé des conseils pertinents. L'accent est mis sur le **processus de décomposition, de raisonnement logique et d'organisation de l'information** pour produire une réponse utile.

**Réponse utilisant la Chaîne de Brouillon :**

**Brouillon 1 (Idées brèves) :** Rome 3 jours : Colisée, Vatican, manger pâtes, pas de monde.

**Brouillon 2 (Tentative d'itinéraire) :** Jour 1 : Colisée. Jour 2 : Vatican. Jour 3 : Autres choses. Mangez de bonnes pâtes. Allez-y tôt.

**Brouillon 3 (Itinéraire plus détaillé et quelques conseils) :**

- **Jour 1 :** Visitez le Colisée et le Forum Romain. Pour manger, essayez une pizzeria.
- **Jour 2 :** Le Vatican est important. Réservez vos billets.
- **Jour 3 :** Vous pouvez voir d'autres sites comme la Fontaine de Trevi. Le matin, il y a moins de monde.

**Brouillon 4 (Version finale - itinéraire structuré, suggestions de restaurants et conseils anti-foules plus complets) :**

Voici une proposition d'itinéraire pour votre séjour de 3 jours à Rome :

- **Jour 1 :** Rome Antique. Commencez par le majestueux Colisée, puis explorez le Forum Romain et le Palatin, tous situés à proximité. Pour le déjeuner, installez-vous dans une trattoria authentique du quartier de Monti et goûtez à des pâtes typiques comme la carbonara ou l'amatriciana. En fin d'après-midi, rejoignez le centre historique pour découvrir le Panthéon, puis dirigez-vous vers la Fontaine de Trevi. La visiter en fin de journée permet de profiter d'une lumière dorée magnifique et d'une atmosphère particulièrement photogénique, malgré une affluence parfois un peu plus élevée. Terminez la journée à la Place d'Espagne, idéalement au coucher du soleil.
- **Jour 2 :** Le Vatican. Consacrez la journée à la Cité du Vatican : la Basilique Saint-Pierre, les Musées du Vatican et l'époustouflante Chapelle Sixtine. Réservez vos billets en ligne à l'avance pour gagner du temps. Pour le dîner, cherchez un restaurant convivial dans le quartier de Borgo, près du Vatican.
- **Jour 3 :** Trastevere et Panthéon. Explorez le charmant quartier de Trastevere avec ses ruelles pittoresques et ses nombreuses options de restauration (essayez une pizza romaine). Visitez ensuite le Panthéon, un chef-d'œuvre architectural. Conseils pour éviter les foules : Visitez les sites tôt le matin, évitez les week-ends, et envisagez de vous déplacer en transports en commun ou à pied.

**Explication de la Chaîne de Brouillon :**

L'IA a commencé par des idées très rudimentaires et a progressivement construit un itinéraire plus détaillé et informatif. Chaque brouillon a ajouté des éléments clés (noms de sites, suggestions de repas, conseils), amélioré la structure et la clarté du texte. L'accent est mis sur l'évolution progressive du contenu et de la présentation pour répondre de manière plus complète et conviviale à la demande de l'utilisateur.

**Différences clés dans ce contexte complexe :**

- **Chaîne de pensée :** Se concentre sur la décomposition logique du problème de planification, l'application de connaissances spécifiques et l'organisation des informations de manière structurée.
- **Chaîne de brouillon :** Se concentre sur la génération itérative du plan de voyage, en commençant par des points basiques et en ajoutant progressivement des détails, des suggestions et un format plus agréable pour l'utilisateur.

Dans des tâches complexes comme la planification de voyage, une combinaison des deux approches

est souvent la plus efficace. L'IA pourrait d'abord utiliser une « Chaîne de Pensée » pour établir une structure logique et identifier les éléments clés, puis utiliser une « Chaîne de Brouillon » pour rédiger et affiner la présentation de l'itinéraire et des conseils.

### 7.2.3 Efficacité, utilisation et limites de la Chain-of-Draft (CoD)

Les premiers résultats scientifiques montrent que *CoD* peut atteindre une précision comparable, voire supérieure, au **CoT** standard sur divers problèmes, tout en consommant moins de tokens. En gros, ça pense presque aussi bien, voire mieux dans certains cas, et plus vite. C'est donc intéressant pour des applications où le coût/*token* est important (production rapide, ou longueur du contexte limitée).

Pour inciter un modèle à faire du *CoD*, il faut calibrer le prompt en ce sens. Donner un exemple de format de brouillon concis, utiliser possiblement un langage plus télégraphique dans la consigne. Par exemple : « *Résous le problème en écrivant le strict minimum d'étapes sous forme de brouillon (n'indique que les chiffres clés ou résultats partiels), puis donne la réponse.* ». On peut aussi préciser « pas plus de X mots par étape ».

Un risque de *CoD* est que, à trop condenser, le modèle pourrait perdre en clarté ou sauter certains détails importants. Il faut trouver le bon équilibre pour que ce qui est omis n'entraîne pas une erreur. *CoD* demande une certaine maîtrise car c'est tenter le modèle de « faire court » – il faut être sûr qu'il ne sacrifie pas la justesse pour la concision.

En résumé, ***Scratchpad et CoD*** sont deux facettes d'une même idée : aider le modèle à mieux gérer les étapes intermédiaires complexes, soit en les externalisant clairement (*scratchpad* détaillé), soit en les externalisant de manière optimisée (brouillons très concis). Dans les deux cas, on **structure la sortie** de manière à ce que le modèle « pense sur le papier » plutôt que tout en interne, ce qui, bien orchestré, améliore ses performances et/ou l'utilité de la réponse pour l'utilisateur.
Après avoir fait réfléchir, agir et noter le modèle, que pourrait-il lui manquer ? Peut-être la capacité de **se relire et de se corriger lui-même**. C'est ce que nous allons voir dans le prochain chapitre consacré à l'auto-vérification et l'auto-amélioration des réponses.

# /Chapitre 8
# L'IA qui s'auto-corrige – Auto-vérification et Auto-raffinement

Même avec de bonnes techniques de *prompting*, un *LLM* peut produire des réponses comportant des erreurs factuelles, des incohérences logiques ou simplement des formulations améliorables. Ne serait-il pas utile que le modèle **prenne du recul** sur sa propre réponse et la corrige avant de nous la présenter ? C'est exactement l'objectif des approches d'**auto-vérification** et d'**auto-raffinement.**

### 8.1 Auto-vérification : la chaîne de vérification (CoVe)

L'**auto-vérification** vise à transformer le *LLM* en son **propre relecteur/contrôleur**. Une méthode proposée est la *Chain-of-Verification (CoVe)*, où le modèle suit un processus en quatre étapes :
Générer une première réponse (comme d'habitude).

**1 -** Générer une série de **questions de vérification** sur cette réponse. C'est comme si le modèle se demandait « Quelles sont les principales choses à vérifier pour s'assurer que ma réponse est correcte ? ». Par exemple, s'il a donné un fait historique, une question de vérification pourrait être « Cette date est-elle exacte ? ».
**2 -** Laisser le modèle répondre à **ses propres questions** de vérification, éventuellement en faisant des recherches (c'est proche de ReAct d'une certaine manière : il peut agir pour vérifier).
**3 -** Sur la base de ces vérifications, produire une **réponse finale révisée**.

Cela permet au modèle de repérer et corriger ses éventuelles **hallucinations factuelles** en se forçant à les confronter à une étape de vérification. Par exemple, un modèle qui initialement répond « Sydney » à « Quelle est la capitale de l'Australie ? » pourrait, en phase de vérification, se poser la question « Sydney est-elle vraiment la capitale de l'Australie ? », réaliser que non (peut-être via son savoir ou une recherche), et du coup corriger en « Canberra » dans sa réponse finale.

En pratique, pour utiliser *CoVe* en *prompt engineering*, on peut inciter le modèle à adopter ce schéma. Par exemple, un *prompt* meta pourrait dire : « Donne d'abord ta réponse, puis vérifie étape par étape chaque assertion, et corrige la réponse si nécessaire. ». Ou on pourrait découper en deux requêtes : une première pour la réponse initiale, puis on repasse le texte en demandant « Vérifie cette réponse et corrige les erreurs. ».

**EXEMPLE** **Question utilisateur :** « Quelle est la capitale de l'Australie ? »

- **Réponse initiale du modèle :** « La capitale de l'Australie est Sydney. » (À ce stade, c'est faux, Sydney n'est pas la capitale.)
- **Phase de vérification (dans le même prompt ou un second prompt) :** Le modèle se repose la question : « La capitale de l'Australie est-elle vraiment Sydney ? », il consulte ses connaissances et se dit « Non, c'est Canberra. ».
- **Réponse finale corrigée :** « Pardon, la capitale de l'Australie est en fait Canberra. »

Ainsi, *CoVe* aurait évité de laisser passer l'erreur initiale en forçant un second regard. Des études ont montré que *CoVe* réduit significativement les **hallucinations factuelles** des modèles sur des tâches de génération de textes informatifs.

**Attention :** L'auto-vérification fonctionne bien si le modèle a les moyens de vérifier. Si l'information n'est pas à sa disposition (pas de connaissance en mémoire ni accès externe), il peut s'auto-convaincre d'une fausse chose. C'est pourquoi coupler *CoVe* avec *RAG* ou *ReAct* (pour aller chercher l'info vraie) est souvent pertinent.

### 8.2 Auto-raffinement : Self-Refine, l'itération critique

Au-delà de vérifier des faits, un modèle peut aussi juger la qualité de sa réponse et tenter de l'améliorer. La technique de *Self-Refine* (auto-raffinement) propose d'utiliser le *LLM* en trois temps : générer une réponse, puis générer une critique de cette réponse, puis générer une nouvelle réponse améliorée tenant compte de la critique. Et éventuellement répéter ce cycle plusieurs fois.

On peut voir *Self-Refine* comme le modèle qui joue à la fois le rôle de l'étudiant et du professeur qui corrige le devoir. Il écrit un premier jet, puis il enfile la casquette du relecteur exigeant pour pointer ce qui ne va pas ou pourrait être mieux, et enfin il réécrit en mieux.

**Ce qu'on peut améliorer :** ça peut être la correction d'erreurs, mais aussi le style, la clarté, la structure, la complétude de la réponse, etc. *Self-Refine* est donc utile même si la première réponse n'est pas fausse : elle peut être simplement perfectible. On obtient souvent une réponse finale de plus haute qualité (plus précise, ou formulée de façon plus élégante, etc.).

**EXEMPLE**

Demande : « Suis les étapes suivantes :

**1 -** Raconte une courte histoire pour enfants mettant en scène un chien et un robot.

**2 -** Relis-toi et propose des axes d'améliorations.

**3 -** Réécris en utilisant les remarques faites au point 2. »

- **Réponse initiale :** Le modèle génère une histoire. Supposons qu'elle soit correcte mais que la fin soit un peu abrupte et qu'il manque une morale.
- **Feedback critique (généré par le modèle sur sa propre histoire) :** « L'histoire est mignonne mais la fin est trop rapide, on ne sait pas bien ce que deviennent le chien et le robot. On pourrait ajouter une morale ou une conclusion plus claire. »
- **Réponse raffinée :** Le modèle réécrit l'histoire en rallongeant la fin, en clarifiant ce que le chien et le robot ont appris de leur aventure, éventuellement en ajoutant une phrase de morale à la toute fin.

On peut répéter : s'il reste un problème, on peut lancer un autre cycle critique->amélioration. Mais en général, un cycle bien paramétré fait déjà un bon boulot. Self-Refine peut s'appliquer à presque tout type de

contenu généré : du code (améliorer l'efficacité, ajouter des commentaires), à la poésie (rendre le style plus fluide), en passant par l'explication (rendre plus pédagogue).

**Prompt engineering pour Self-Refine :** On peut intégrer cela en un seul prompt du style : « Donne une première réponse, puis analyse-la et améliore-la. », mais souvent on préfère séparer les étapes pour garder le contrôle. Par exemple, on demande d'abord la réponse brute. Puis on donne cette réponse au modèle en disant « Peux-tu évaluer cette réponse et proposer mieux ? ». Cela permet aussi potentiellement à un humain de voir la version 1 et la version finale.

**Avantages :** *Self-Refine* permet d'obtenir une qualité qui approcherait une **révision humaine**. Il a été noté que cela améliore la qualité sans nécessiter de données étiquetées supplémentaires (le modèle s'auto-évalue avec ses critères internes). C'est comme une mini itération de *feedback* interne. Dans un contexte où l'on ne veut pas multiplier les appels de modèle, on peut aussi faire une seule requête demandant « Donne ta réponse puis ta version améliorée » en espérant qu'il joue le jeu, mais c'est plus sûr en plusieurs tours.

**Limitations :** Tout modèle n'est pas extrêmement bon critique de lui-même. Parfois, il peut manquer des défauts ou au contraire sur-corriger ce qui n'avait pas besoin de l'être. De plus, chaque itération consomme du temps et des *tokens*. Donc on ne peut pas se permettre 10 itérations. En pratique, 1 cycle de critique/amélioration apporte déjà un gain visible avec un coût modeste.

### 8.3 Combiner CoVe et Self-Refine

Auto-vérification et auto-raffinement peuvent tout à fait se marier. On peut d'abord vérifier factuellement la réponse (CoVe) puis une fois qu'elle est factuellement correcte, on peut affiner le style (Self-Refine). Ou inversement, peu importe l'ordre selon les besoins.

**EXEMPLE** **Vérification Factuelle puis Raffinement Stylistique (CoVe puis Self-Refine)**

**1 -** « Raconte une courte histoire pour enfants mettant en scène un chien et un robot.
**2 -** Relis-toi et propose des axes d'améliorations.
**3 -** Réécris en utilisant les remarques faites au point 2. »
**- Prompt :** *Répondez à la question suivante de manière factuellement correcte, puis rédigez votre réponse sous la forme d'un court paragraphe informatif avec un ton engageant et accessible au grand public.*
**Question :** Quelle est la plus haute montagne d'Afrique ?

[Le fait de demander au modèle de « ***répondre de manière factuellement correcte*** » et ensuite de rédiger un « ***court paragraphe informatif*** » avec un « ***ton engageant et accessible*** » crée une structure dans laquelle :

- La première partie de la tâche (réponse factuelle) oriente le modèle vers un raisonnement basé sur des faits vérifiables.
- La deuxième partie (réponse engageante) indique que le modèle doit présenter ces faits de manière fluide et accessible. Cela peut amener le modèle à vérifier les faits pour s'assurer qu'il ne se trompe pas avant de les expliquer plus clairement dans un format narratif.]

Déroulement implicite (dans l'IA) :

**Auto-Vérification (CoVe) :**

- Générer une réponse initiale (par exemple : « Le Mont Kenya est la plus haute montagne d'Afrique. »).
- Se poser la question : « Est-ce exact ? »
- Effectuer une recherche interne ou externe pour vérifier.
- Découvrir que le Mont Kilimandjaro est en réalité le plus haut.
- Corriger la réponse factuellement : « Le Mont Kilimandjaro est la plus haute montagne d'Afrique. »

**Auto-Raffinement (Self-Refine) :**

**Évaluer la réponse factuellement correcte :**

Le Mont Kilimandjaro est la plus haute montagne d'Afrique. » (Jugée trop brute).

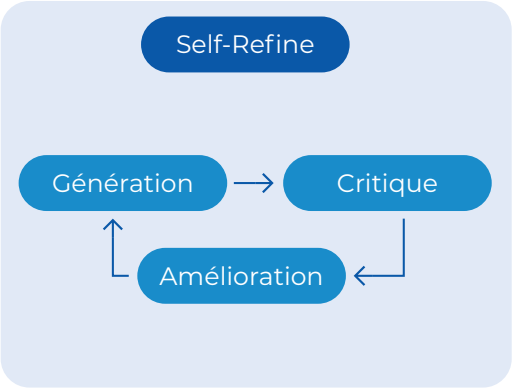
Se poser des questions sur le style : « Comment rendre cela plus engageant ? Quels détails intéressants puis-je ajouter ? »

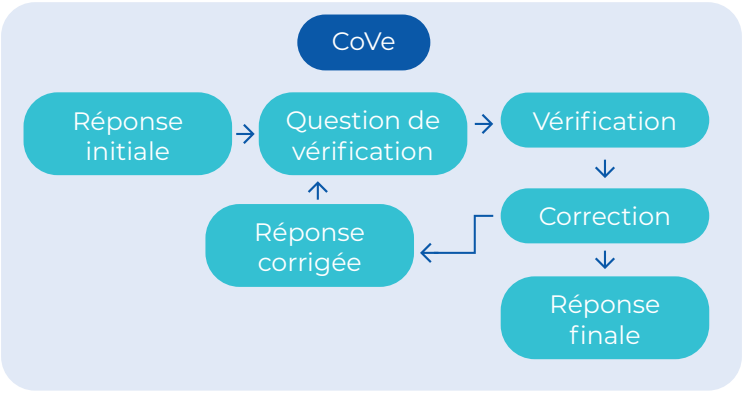
**Raffiner le style et ajouter des détails :**

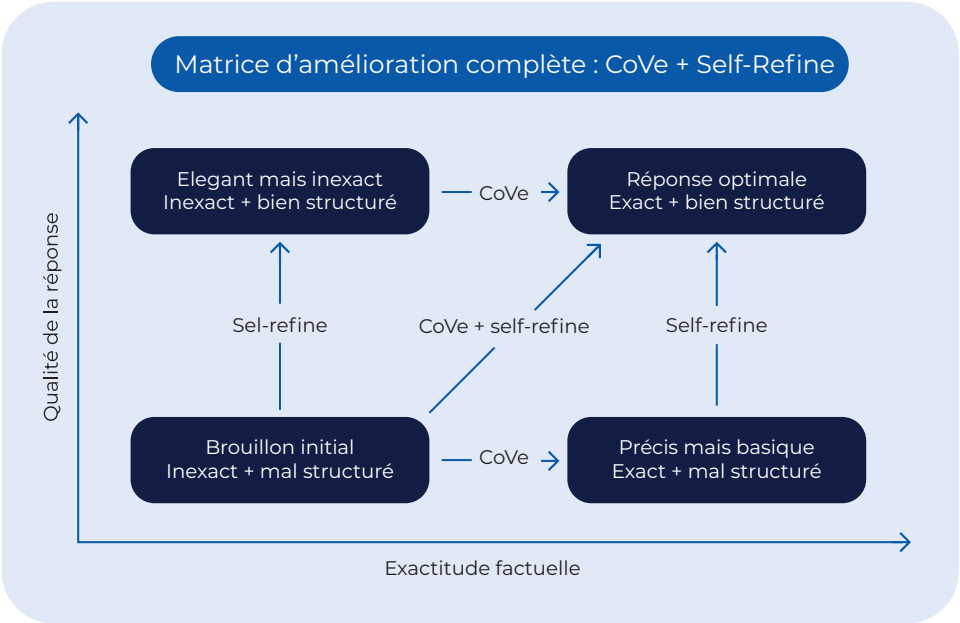
S'élevant majestueusement au-dessus des plaines de Tanzanie, le Kilimandjaro n'est pas seulement la plus haute montagne d'Afrique, culminant à environ 5 895 mètres d'altitude. Son sommet enneigé, souvent coiffé de nuages, offre un spectacle à couper le souffle et représente un défi emblématique pour les alpinistes du monde entier.

**Explication :** Le *prompt* guide l'IA à d'abord s'assurer de l'exactitude de l'information *(CoVe)* avant de se concentrer sur la manière de la présenter de façon attrayante *(Self-Refine)*.

L'important est que l'on dispose là de techniques pour **amener le modèle à revenir sur sa copie** et à l'améliorer, plutôt que d'accepter la première sortie brute. En *prompt engineering*, cela signifie penser éventuellement en **plusieurs étapes** : on ne pose pas juste la question finale, on peut orchestrer une petite séquence « question -> réponse -> vérification -> amélioration -> réponse finale ». Bien sûr, cela allonge le processus, mais pour des tâches critiques (par ex., générer un résumé fiable d'un document légal, ou écrire du code où une erreur peut coûter cher), ça vaut le coup.




**En résumé de ce chapitre,** nous avons vu comment un *LLM* peut être guidé pour **évaluer et perfectionner** ses propres réponses. L'auto-vérification *(CoVe)* le transforme en examinateur méthodique qui traque ses erreurs factuelles, tandis que l'auto-raffinement *(Self-Refine)* le fait devenir son propre coach d'écriture, améliorant formulation et contenu. Ces approches témoignent d'une chose : on peut exploiter les capacités latentes d'un grand modèle non seulement pour produire du texte, mais aussi pour juger ce texte.

En mobilisant ce regard critique interne, on franchit une étape supplémentaire vers des réponses plus fiables et de meilleure qualité, sans intervention humaine directe dans la boucle.

# / Conclusion et perspectives

Au fil de ce livre blanc, nous avons parcouru un chemin riche partant des **fondamentaux du *prompt engineering*** jusqu'aux techniques avancées de pointe, à la date de mai 2025. Nous avons appris à parler clairement aux *LLM*, à leur donner du contexte, un rôle, un format, ou des exemples (chapitres 1 et 2). Puis nous avons découvert comment aller plus loin en exploitant leurs capacités émergentes de raisonnement complexe.

Le dénominateur commun de toutes ces techniques est d'**orienter et d'optimiser le comportement** du *LLM* sans avoir à en modifier les paramètres, simplement en trouvant le bon ingénieur *prompt* – c'est-à-dire vous – qui sait quelles instructions ou structures donner. Le prompt engineering apparaît à la fois comme un art (il faut de la créativité, de l'intuition, parfois de l'essai-erreur) et comme une science (les techniques sont testées, documentées, on s'appuie de plus en plus sur des travaux académiques solides).

**Quelles sont les prochaines étapes ?** Le domaine évolue incroyablement vite. Voici quelques tendances et conseils pour l'avenir :

- **Combinaison des techniques :** Les méthodes que nous avons vues ne sont pas mutuellement exclusives. On peut – et on va de plus en plus – les combiner dans un même système. Par exemple, un agent conversationnel peut utiliser ReAct pour chercher des infos, puis *CoT* pour raisonner sur ces infos, puis Self-Refine pour peaufiner sa réponse. En tant que prompt engineer, pensez modulaires : chaque problème peut nécessiter d'activer plusieurs « outils » dans votre prompt toolbox.
- **Automatisation du prompt engineering :** Un comble ? Eh bien oui, il existe des recherches sur l'*Auto-Prompting* – comment utiliser l'IA pour générer ou améliorer automatiquement les prompts. Des techniques comme APE (Automatic Prompt Engineering) essayent par exemple de faire proposer par le modèle lui-même des variantes de prompt et de tester celles qui marchent le mieux. L'objectif est d'alléger le travail humain pour trouver le prompt idéal. Nous ne sommes pas encore au point où l'IA remplace totalement le prompt engineer, mais il est possible que dans le futur, certaines de ces tâches soient en partie automatisées.

- **LLM de plus en plus interactifs et contextualisés :** À mesure que ces modèles seront intégrés dans des produits, ils seront dotés de « *plugins* » ou d'extensions leur permettant d'accéder à Internet, d'interagir avec des applications, etc. Le rôle du prompt (ou du système qui construit dynamiquement le prompt) sera crucial pour orchestrer ces interactions en toute sécurité. ReAct en est un avant-goût.

- **Gardez l'éthique et la sécurité en tête :** Plus les *prompts* deviennent complexes et puissants, plus on doit faire attention à leur usage. Par exemple, donner accès à Internet à un *LLM* peut lui faire lire tout et n'importe quoi, il faudra donc des garde-fous (le prompt peut contenir des instructions de filtrage). L'auto-correction du modèle peut être utilisée aussi pour éviter des dérives (on peut lui

demander « Assure-toi que ta réponse ne viole aucune règle de contenu avant de la donner »). Le prompt engineer de demain devra travailler main dans la main avec des spécialistes en éthique de l'IA pour s'assurer que ces modèles restent alignés avec nos valeurs.

En conclusion, le ***prompt engineering*** s'affirme comme une compétence incontournable pour quiconque souhaite tirer le meilleur des intelligences artificielles conversationnelles et génératives. C'est un domaine jeune, en évolution rapide, et ce livre blanc vous a présenté un instantané des connaissances actuelles et des techniques émergentes au cours de l'année 2025. Armez-vous de ces concepts, expérimentez-les, amusez-vous à **innover** vous-même en les combinant, et surtout gardez l'esprit ouvert : chaque avancée des modèles ouvre la porte à de nouvelles idées de prompts.

Le voyage ne fait que commencer, et vous êtes désormais bien équipé pour, à votre tour, **pousser plus loin l'art du *prompt engineering***. À vous de jouer

# / Index des schémas

# / Glossaire

## A

**Affinage (Fine-tuning) :** Processus d'optimisation d'un modèle de langage pré-entraîné pour des tâches spécifiques en l'entraînant sur des données ciblées. Cette étape permet de spécialiser le modèle après son apprentissage général sur de vastes corpus de textes.

**Agent conversationnel :** Système d'IA utilisant des LLM pour maintenir des dialogues cohérents avec les utilisateurs en intégrant diverses techniques de prompt engineering. Il peut combiner raisonnement, actions externes et mémorisation du contexte.

**Architecture Transformer :** Structure de réseau de neurones révolutionnaire basée sur le mécanisme d'attention, permettant aux LLM de traiter tous les tokens d'une phrase simultanément. C'est l'architecture fondamentale qui propulse la plupart des grands modèles de langage modernes.

**Arbre de Pensées (Tree-of-Thoughts, ToT) :** Technique avancée permettant au modèle d'explorer plusieurs pistes de raisonnement en parallèle plutôt qu'un chemin linéaire. Le modèle peut ainsi bifurquer, évaluer différentes options et choisir les branches les plus prometteuses.

**Auto-cohérence (Self-Consistency) :** Méthode qui génère plusieurs chaînes de pensée indépendantes pour la même question, puis utilise un vote majoritaire pour déterminer la réponse finale. Cette approche améliore la fiabilité en réduisant les erreurs de raisonnement isolées.

**Auto-raffinement (Self-Refine) :** Processus en trois étapes où le modèle génère une réponse, produit une critique de cette réponse, puis génère une version améliorée. Cette technique permet d'améliorer la qualité, le style et la précision des réponses.

**Auto-vérification** : Capacité du modèle à examiner et corriger ses propres réponses avant de les présenter à l'utilisateur. Cette approche réduit les hallucinations et améliore la fiabilité des informations générées.

## C

**CCRFE (Ce Chat Rigole Fort Evidemment) :** Mnémotechnique pour retenir les 5 piliers du prompt engineering efficace : Clarté, Contexte, Rôle, Format, Exemples. Ces éléments constituent les fondements d'un prompt réussi.

**Chain-of-Draft (CoD) :** Évolution du scratchpad visant à rendre le processus de raisonnement moins coûteux en tokens en produisant des brouillons minimalistes. Le modèle génère des étapes concises contenant uniquement l'information cruciale pour avancer vers la solution.

**Chaîne de Pensée (Chain-of-Thought, CoT) :** Technique fondamentale qui incite le modèle à expliciter son raisonnement étape par étape avant de donner sa réponse finale. Cette approche améliore significativement les performances sur les tâches nécessitant un raisonnement complexe.

**Chain-of-Verification (CoVe) :** Processus d'auto-vérification en quatre étapes : génération d'une réponse, création de questions de vérification, réponse à ces questions, puis production d'une réponse finale corrigée. Cette méthode réduit les hallucinations factuelles.

**Contexte :** Ensemble des informations pertinentes fournies dans un prompt pour guider l'attention du modèle vers la réponse souhaitée. Un contexte riche et approprié est essentiel pour obtenir des réponses précises et adaptées.

## D

**Décodage déterministe :** Stratégie de sélection qui choisit systématiquement le token ayant la probabilité la plus élevée lors de la génération. Cette approche produit des textes cohérents mais prévisibles, idéale pour les tâches factuelles.

**Décodage stochastique :** Méthode de sélection qui tire au sort le prochain token en respectant la distribution de probabilités calculée. Cette approche introduit de la variété et de la créativité dans la génération, contrôlée par le paramètre de température.

## E

**Embeddings :** Représentations vectorielles numériques des tokens ou mots permettant aux modèles de traiter le langage mathématiquement. Ces vecteurs capturent les relations sémantiques entre les éléments linguistiques.

## F

**Fenêtre de contexte :** Quantité maximale de texte (mesurée en tokens) qu'un modèle peut prendre en compte simultanément pour générer une réponse. Cette limitation détermine la «mémoire à court terme» du modèle.

**Few-shot learning :** Technique d'apprentissage par quelques exemples où l'on inclut dans le prompt plusieurs exemples complets (entrées et sorties attendues). Le modèle apprend par imitation à résoudre la nouvelle tâche en suivant le modèle des exemples fournis.

**Format de sortie :** Spécification claire de la structure et de la présentation attendues pour la réponse du modèle. Définir le format permet d'obtenir des réponses bien structurées et facilement exploitables.

## G

**Graphe de Pensées (Graph-of-Thoughts, GoT) :** Extension conceptuelle de l'Arbre de Pensées où les chemins de raisonnement forment un graphe plutôt qu'une hiérarchie arborescente. Cette approche permet des connexions plus flexibles entre les idées et la réutilisation de fragments de raisonnement.

## H

**Hallucination :** Phénomène où le modèle génère des informations qui semblent plausibles mais sont factuellement incorrectes ou non fidèles à la réalité. Ces erreurs résultent souvent d'un manque d'informations fiables dans le contexte fourni.

## I

**Inférence :** Phase d'utilisation d'un modèle pré-entraîné pour générer des réponses à partir de nouveaux prompts. C'est le processus de génération de texte token par token basé sur les probabilités calculées.

**Instruction directe :** Type de prompt le plus basique consistant en une demande claire et concise pour réaliser une tâche spécifique. Cette approche convient aux tâches simples et bien définies.

## L

**Least-to-Most (LtM) :** Méthode de décomposition progressive qui résout d'abord des versions simplifiées d'un problème avant de traiter la version complète. Cette approche pédagogique aide le modèle à aborder des tâches complexes par étapes croissantes de difficulté.

**LLM (Large Language Model) :** Modèle de langage de grande taille entraîné sur d'énormes quantités de textes pour comprendre et générer du langage naturel. Ces modèles utilisent généralement l'architecture Transformer et comptent des milliards de paramètres.

**Lost in the middle :** Phénomène où les modèles ont tendance à mieux traiter les informations situées au début et à la fin d'un long contexte, au détriment du milieu. Ce biais pose des défis pour le traitement de très longs documents.

**M**

**Mécanisme d'attention :** Composant clé des Transformers qui permet de calculer l'importance et les relations entre tous les tokens d'une séquence simultanément. Ce mécanisme permet au modèle de «porter attention» aux éléments les plus pertinents pour comprendre chaque token.

**P**

**Paramètres :** Voir Poids du modèle. Valeurs numériques ajustées pendant l'entraînement qui encodent les connaissances et capacités du modèle.

**Persona :** Rôle attribué au modèle dans un prompt pour orienter son style de réponse et mobiliser des connaissances spécialisées. Par exemple, «réponds comme un professeur d'histoire» ou «adopte le rôle d'un expert en marketing».

**Poids du modèle :** Paramètres internes numériques du modèle qui encodent ses connaissances sur le langage, les relations entre tokens et les règles linguistiques. Ces valeurs sont ajustées automatiquement pendant l'entraînement pour minimiser les erreurs de prédiction.

**Pré-entraînement :** Phase initiale d'apprentissage où le modèle est exposé à d'énormes quantités de textes bruts pour apprendre à prédire le token suivant. Cette étape permet au modèle d'acquérir une compréhension générale du langage et du monde.

**Prompt :** Instruction, question ou texte de départ fourni à un modèle de langage pour orienter sa réponse. Un prompt efficace combine clarté, contexte, rôle, format et exemples pour maximiser la qualité de la génération.

**Prompt contextuel :** Type de prompt qui enrichit une demande simple en fournissant un contexte supplémentaire pour guider le modèle. Ce contexte peut inclure des informations sur le sujet, le but, le public cible ou les contraintes spécifiques.

**Prompt data-driven :** Prompt qui fournit des données brutes au modèle et lui demande de les analyser, transformer ou en extraire des informations. Cette approche est utilisée pour des tâches d'analyse de données textuelles, de classification ou de résumé.

**Prompt engineering :** Art et science de concevoir des instructions optimales pour maximiser les performances des modèles de langage. Cette discipline combine techniques linguistiques, logique et stratégie pour obtenir des réponses de haute qualité.

**Prompt instructif :** Type de prompt fournissant des instructions très détaillées et précises avec plusieurs contraintes ou directives spécifiques. Cette approche permet un contrôle fin sur la génération et convient aux tâches complexes nécessitant des spécifications précises.

**R**

**RAG (Retrieval-Augmented Generation) :** Architecture en deux étapes qui récupère d'abord des documents pertinents dans une base de connaissances, puis les intègre dans le prompt pour enrichir la génération. Cette approche réduit les hallucinations en fournissant au modèle des informations factuelles vérifiables.

**ReAct (Reason + Act) :** Framework combinant raisonnement interne et actions externes en cycles alternés de Pensée → Action → Observation. Cette approche transforme le LLM d'un simple générateur de texte en un agent capable d'interagir avec son environnement.

**Rôle :** Synonyme de Persona. Identité ou perspective spécifique attribuée au modèle dans un prompt pour orienter sa réponse vers un domaine d'expertise particulier. Définir un rôle permet d'exploiter les connaissances spécialisées du modèle et d'adapter le ton de la réponse.

**S**

**Scratchpad :** Espace de travail où le modèle peut noter ses calculs, variables et pensées intermédiaires avant de donner sa réponse finale. Cette technique aide le modèle à structurer sa réflexion et à ne pas perdre de vue les détails importants lors de raisonnements complexes.

**Self-Consistency :** Voir Auto-cohérence.

**Self-Refine :** Voir Auto-raffinement.

**T**

**Température :** Paramètre contrôlant le degré de créativité et de prise de risque lors de la sélection des tokens. Une température basse (0-0.3) produit des réponses prévisibles, tandis qu'une température élevée (0.7-1.0+) encourage la créativité au risque d'incohérences.

**Token :** Unité de base du traitement linguistique par les LLM, pouvant représenter un mot entier, une partie de mot ou un caractère spécial. Ces «briques LEGO du langage» permettent aux modèles de travailler avec un vocabulaire limité tout en traitant un langage potentiellement illimité.

**Tokenisation :** Processus de découpage d'un texte en séquence de tokens par un outil logiciel appelé tokenizer. Cette transformation permet aux modèles de traiter le langage humain sous forme numérique.

**Top-K :** Paramètre qui limite la sélection des tokens aux K candidats les plus probables lors de la génération. Des valeurs basses (K=20-40) produisent des réponses plus conservatrices, tandis que des valeurs élevées (K>50) permettent plus de variété.

**Top-P (nucleus sampling) :** Méthode de sélection qui ne considère que les tokens dont la probabilité cumulée atteint un seuil P. Cette approche offre un contrôle plus fin que Top-K en s'adaptant dynamiquement au nombre de candidats viables.

**Transformer :** Architecture de réseau de neurones révolutionnaire utilisant le mécanisme d'attention pour traiter le langage. Cette structure permet aux modèles de comprendre les relations contextuelles entre tous les éléments d'une séquence simultanément.

**Tree-of-Thoughts (ToT) :** Voir Arbre de Pensées.

**Zero-shot :** Capacité d'un modèle à effectuer une tâche sans exemple préalable, uniquement grâce à des instructions claires. Cette approche teste la compréhension générale du modèle et sa capacité de généralisation.

**Z**

**Zero-shot CoT :** Application de la chaîne de pensée sans exemples préalables, généralement déclenchée par une instruction simple comme «Réfléchissons étape par étape». Cette technique active le mode de raisonnement séquentiel dans les grands modèles récents.

Et si parler à une IA devenait une compétence aussi essentielle que savoir écrire ou coder ? L'intelligence artificielle redéfinit en profondeur notre rapport au numérique. Pour en tirer pleinement parti, une nouvelle compétence s'impose : le prompt engineering. Bien plus qu'un simple art de converser avec les modèles de langage, il s'impose aujourd'hui comme une discipline à part entière, à la croisée du langage, de la logique et de la stratégie.

Issu de l'expérience de deux experts de la transformation numérique, cet ouvrage propose une méthode claire, progressive et immédiatement applicable. Vous y trouverez des clés pour rédiger des prompts efficaces, comprendre les mécanismes des grands modèles de langage, et exploiter des techniques avancées comme la Chaîne de Pensée, la méthode Least-to-Most ou les frameworks ReAct et RAG.

Conçu pour les professionnels, les enseignants, les étudiants ou les curieux passionnés, ce livre ne se contente pas de présenter des outils : il propose une démarche intellectuelle pour mobiliser le potentiel de l'IA de manière plus fine, plus pertinente, plus éclairée.
Car l'avenir n'appartiendra pas à ceux qui utilisent l'IA, mais à ceux qui sauront lui parler avec justesse.