Prof. Imane Allaouzi

TP : Ingénierie de prompt appliquée à la génération de code avec l'IA

Contexte :

Autrefois, le développement logiciel reposait essentiellement sur des processus manuels, mobilisant beaucoup de temps et de ressources humaines à chaque étape, de l'analyse des besoins jusqu'au déploiement.

Aujourd'hui, l'IA a remodelé ces méthodes. Intégrés de façon fluide dans presque toutes les phases du cycle de développement, les outils et techniques basés sur l'IA permettent aux équipes de concevoir, coder et livrer plus rapidement, avec une efficacité et une précision accrues.

Parmi les avancées les plus prometteuses figure l'IA générative. Elle ne se limite plus à de simples outils : elle devient un véritable copilote, capable d'influencer directement la production de code.

« Le futur du développement ne repose pas sur l'absence de code, mais sur moins de code écrit manuellement, et plus d'intelligence dans la manière de le générer. »

Aujourd'hui, un développeur n'est plus uniquement celui qui écrit du code, mais aussi celui qui sait collaborer efficacement avec des modèles d'IA pour le générer, le tester et l'améliorer.

Objectif :

Comprendre comment concevoir des prompts efficaces pour générer du code de qualité à l'aide d'IA génératives, tout en appliquant les principes fondamentaux : **Tâche, Contexte, Références, Évaluation, Itération**.

Partie 1 : Choix de la Solution d'IA Générative

Pour réaliser ce TP, chaque étudiant devra choisir une seule des solutions d'IA générative "Code". Votre choix sera un élément central de votre analyse.

Avant de commencer les exercices du TP, vous devrez remplir la section suivante :

1. Solution choisie : Nom de la solution choisie par l'étudiant/groupe

2. Définition brève de la solution : Décrivez-en une ou deux phrases ce qu'est cet outil/API et sa fonction principale.

3. Avantages perçus de cette solution pour le développement de code : Listez au moins 3 avantages que vous anticipez ou avez déjà observés.

4. Inconvénients ou limites perçues de cette solution : Listez au moins 3 inconvénients ou limites que vous anticipez ou avez déjà observés.

5. Cas d'utilisation typiques : Décrivez brièvement dans quels scénarios de développement cet outil est particulièrement pertinent.

Partie 2 – Génération de code avec IA

Objectif :

Montrer comment un bon prompt permet d'obtenir un code plus précis, clair et adapté.

Exercice 2.1 :

Générer une fonction nommée **calculate** qui accepte deux entiers et une chaîne représentant une opération (add, subtract, multiply, divide), et retourne le résultat de l'opération.

Exemples d'Entrées/Sorties attendues :

```
calculate(5, 3, '+') → 8
calculate(5, 3, '-') → 2
calculate(5, 3, '*') → 15
calculate(5, 3, '/') → 1.67
```

Étapes d'expérimentation :

- ***Prompt Vague :***

"*Écris une fonction pour faire des opérations entre deux nombres en Python.*"

Questions : La fonction est-elle nommée ? Quelles opérations sont prises en charge ? Y a-t-il une gestion des erreurs ? Des commentaires ?

- ***Prompt Spécifique :***

"*Écris une fonction Python appelée calculate(a, b, op) qui prend deux entiers a et b, et une chaîne op indiquant '+', '-', '*', ou '/'. La fonction doit retourner le résultat de l'opération, gérer les erreurs (division par zéro, opération invalide) et arrondir le résultat de la division à deux décimales. Ajoute un docstring détaillé et des commentaires.*"

Questions : Comparer la robustesse, la lisibilité et la couverture des cas.

- ***Prompt avec Persona :***

"*En tant que développeur Python, écris une fonction calculate(a, b, op) qui prend deux entiers et une chaîne représentant une opération mathématique ('+', '-', '*', '/'). La fonction doit être robuste, bien documentée, gérer les erreurs (division par zéro, opérateur invalide), arrondir les divisions à deux décimales, et respecter les conventions PEP8. Inclue un docstring et des commentaires clairs.*"

Questions : Le code est-il plus professionnel ? Mieux structuré ? Plus sécurisé ?

Tests :

Question : Pour chaque version générée, demandez à l'IA de générer 3-4 tests unitaires pytest pertinents et exécutez-les.

Analyse Critique (dans README.md) :

1) Décrivez les différences observées entre les codes générés par chaque prompt.
2) Quel principe de Prompt Engineering (clarté, spécificité, persona) a eu le plus grand impact et pourquoi ?
3) L'IA a-t-elle introduit des erreurs ou des comportements inattendus dans une des versions ?
4) Quel est le coût (en temps et en effort) pour obtenir un code de haute qualité avec un prompt vague vs. un prompt spécifique ?

Exercice 2.2 :

Vous travaillez sur une application où les chaînes de caractères suivent un format spécifique que vous avez défini en interne.

Générer une fonction Python **format_product_code(product_id)** qui prend un product_id (chaîne) et le formate selon une règle complexe.

Étapes d'expérimentation :

Prompt basé sur la Règle (zéro-Shot Prompting) :

"*Crée une fonction Python format_product_code(product_id). Le product_id doit être une chaîne de 10 caractères alphanumériques. La fonction doit insérer un tiret après le 3ème et le 7ème caractère. Si la chaîne n'a pas 10 caractères ou contient des caractères non alphanumériques, elle doit lever une ValueError. Inclue un docstring.*"

Question : Le code est-il correct ? Est-il robuste face aux erreurs ?

Prompt avec un Exemple (One-Shot Prompting) :

"*Crée une fonction Python format_product_code(product_id). Le product_id doit être une chaîne de 10 caractères alphanumériques. Voici un exemple d'entrée-sortie: format_product_code("ABC123DEF4") devrait retourner "ABC-123-DEF4". La fonction doit lever une ValueError si l'entrée est invalide. Inclue un docstring.*"

Question : Comparer avec le résultat précédent. L'exemple a-t-il simplifié la tâche de l'IA ? A-t-il aidé à éviter des erreurs ?

Prompt avec Multiple Exemples (Few-Shot prompting) :

Reprenez le prompt précédent et ajoutez un deuxième exemple *d'entrée-sortie:* format_product_code("XYZ987GHIJ") *devrait retourner* "XYZ-987-GHIJ", incluant un cas d'erreur : format_product_code("SHORT") devrait lever une ValueError.

Question : L'IA gère-t-elle mieux les cas d'erreur maintenant ? La robustesse a-t-elle été améliorée ?

Tests : Générer des tests unitaires pytest pour chaque version et les exécuter.

Analyse Critique (dans README.md) :

1) Analysez comment l'ajout d'exemples a influencé la capacité de l'IA à comprendre et à générer le code correct, en particulier pour les règles complexes ou les gestions d'erreurs.
2) Quand le "Few-Shot Prompting" est-il particulièrement utile en développement ?

3) Y a-t-il des limites aux exemples (nombre, qualité) ?

Exercice 2.3 :

Rédiger un prompt pour générer une mini-application Web en HTML/CSS/JS qui simule une calculatrice simple.

Étapes d'expérimentation :

1) Rédiger un prompt vague → observer le résultat.
2) Ajouter des détails techniques (styles, types de boutons, gestion des erreurs).
3) Évaluer les différences.

Partie 3 – Débogage et Amélioration du Code

Objectif :

Utiliser l'IA pour corriger des bugs, refactoriser et documenter du code existant.

Execrice 3.1 : Débogage assisté

Vous recevez un code d'un collègue qui présente des erreurs et n'est pas optimal.

```
def calculate_average(numbers_list):
    # This function calculates the average of numbers in a list
    # It has some issues
    total = 0
    for num in numbers_list:
        total += num
    average = total / len(numbers_list)
    return average


# Example of usage (might cause errors)
my_nums = [1, 2, 'three', 4] # <-- Error here
avg = calculate_average(my_nums)
print(f"Average: {avg}")
```

Question :

1) Exécutez le code et observez l'erreur (TypeError).
2) Fournissez le code *calculate_average,* le message d'erreur complet (traceback) et demandez à l'IA d'identifier la cause de l'erreur et de proposer une correction.
3) Demandez à l'IA de générer des tests unitaires avec pytest pour valider son comportement.

Execrice 3.2 : Refactoring avec l'IA

Le code ci_dessous trie une liste d'entiers, mais il est :

- non structuré,
- difficile à lire (noms non explicites),
- sans fonction ni commentaires.

Exemple de code de départ à refactorer :

```
# code de départ
a = [5, 3, 8, 6, 7, 2]
for i in range(len(a)):
    for j in range(i+1, len(a)):
        if a[i] > a[j]:
            tmp = a[i]
            a[i] = a[j]
            a[j] = tmp
print(a)
```

Question :

1) Analyser le code fourni (quelle est sa fonction ? ses défauts de lisibilité ?).
2) Formuler un prompt de refactoring clair.
3) Ajouter des contraintes :

- Suivre la convention **PEP8**,
- Ajouter des **docstrings**,
- Séparer en **fonctions modulaires**,
- Utiliser des **noms explicites** pour les variables et les fonctions,
- Ajouter un **bloc if __name__ == "__main__"**.

Execrice 3.3 : Génération de Documentation

Vous avez une fonction clé dans votre projet, mais elle manque de documentation claire pour les futurs contributeurs.

```
def get_user_permissions(user_id, system_context):
    # This function fetches user permissions
    # Needs better documentation
    if user_id in system_context['admins']:
        return ['read', 'write', 'delete', 'admin']
    elif user_id in system_context['editors']:
        return ['read', 'write']
    else:
        return ['read']
```

Question :

1) Demandez à l'IA de générer un docstring complet et conforme à un standard pour la fonction get_user_permissions. Le docstring doit décrire : le but de la fonction, ses arguments (user_id, system_context), sa valeur de retour, et un exemple d'utilisation.

2) Demandez à l'IA de générer une section Markdown pour le fichier README.md du projet, expliquant comment utiliser cette fonction, ses prérequis (le format de system_context), et des exemples d'appel.

3) Lisez le docstring et la section README générés. Sont-ils clairs ? Complets ? Faciles à comprendre pour un autre développeur ?