Rapport Final : Ingénierie des Prompts pour la Génération de Code par IA

Auteur: elabdellaoui moad

Classe: GI1

CNE: s135335757

Professeur: Prof. Imane Allaouzi

Date: 14/06/2025

1. Introduction

La discipline du développement logiciel est au coeur d'une transformation profonde, passant de processus historiquement manuels à une nouvelle ère de création augmentée par l'IA. L'IA générative s'est imposée à l'avant-garde de ce changement, s'établissant non pas comme un simple outil auxiliaire, mais comme un véritable "copilote" pour les développeurs, capable d'influencer et d'accélérer directement la production de code de haute qualité. Cette évolution concrétise la vision d'un avenir où le développement est défini "non pas par une absence de code, mais par moins de code écrit manuellement, et plus d'intelligence dans la manière de le générer".

Ce rapport fournit une analyse complète et détaillée des résultats d'un travail pratique sur l'ingénierie des prompts. Son objectif est de documenter et d'évaluer méticuleusement comment différentes techniques de conception de prompts - allant des requêtes vagues aux spécifications détaillées et basées sur des personas - impactent directement la qualité, la robustesse et le professionnalisme du code généré par l'IA. Les conclusions sont basées sur une série d'expériences structurées en génération de code, débogage, refactoring et documentation.

2. Analyse de la Solution d'IA Générative Choisie : Gemini

La solution d'IA sélectionnée pour ce travail pratique était Gemini, la famille de grands modèles de langage multimodaux développée par Google. Mon évaluation initiale était basée sur ses capacités décrites, que j'ai soupesées avant de commencer les exercices.

2.1.Avantages Initiaux Perçus

Sur la base de la description de l'outil, j'anticipais les avantages suivants pour les tâches de développement logiciel :

- Capacités Multimodales Intuitives : La possibilité de soumettre des requêtes complexes mêlant du code à des captures d'écran d'interfaces utilisateur ou des schémas d'architecture pour une résolution de problèmes plus contextuelle.
- Analyse de Code Complexe : La capacité de traiter et d'analyser de grands blocs de code pour comprendre la logique, identifier les dépendances et maintenir le contexte, ce qui le rend idéal pour le débogage et le refactoring.
- Accès à l'Information en Temps Réel : L'intégration avec Google Search, garantissant que le code généré utiliserait les bibliothèques les plus récentes et respecterait les normes de programmation actuelles.

2.2.Inconvénients Initiaux Perçus

J'ai également identifié plusieurs limitations potentielles qui nécessiteraient une gestion attentive :
- Risque d'"Hallucinations" : Comme toute IA, Gemini peut produire des réponses incorrectes ou du code contenant des bogues subtils, rendant la vérification et la validation humaines indispensables.
- Perte de Contexte Conversationnel : Dans des échanges très longs, le modèle pourrait "oublier" des détails ou des contraintes mentionnés précédemment, conduisant à des suggestions qui s'écartent de l'objectif initial.
- Dépendance et Impact sur l'Apprentissage : Une dépendance excessive à l'outil pourrait freiner le développement personnel des compétences en résolution de problèmes et créer une dépendance même pour les tâches de base.

2.3.Réflexion Post-TP

Les exercices pratiques ont fourni une perspective beaucoup plus claire et fondée sur des preuves des capacités de Gemini. Mon évaluation initiale était largement exacte, mais les implications réelles de ses forces et de ses faiblesses sont maintenant plus apparentes.
L'avantage de sa puissance analytique a été confirmé à plusieurs reprises. Lorsqu'on lui a fourni des prompts précis et techniques, Gemini a produit un code exceptionnel, comme on l'a vu dans l'exercice de refactoring (3.2) où il a transformé un simple script en un programme modulaire et professionnel. Sa capacité à générer une documentation de haute qualité (3.3) était également

remarquable.

Cependant, la limitation concernant l'interprétation du contexte et le biais des exemples s'est avérée très significative. L'analyse de l'Exercice 2.2 était particulièrement révélatrice :

"L'IA a donc tendance à donner plus de poids aux exemples concrets qu'aux règles abstraites, ce qui peut simplifier la tâche mais aussi la rendre moins complète si les exemples ne couvrent pas tous les cas."

Cette tendance à "sur-apprendre" d'un ensemble incomplet d'exemples a entraîné une régression tangible de la qualité du code. Ma conclusion est que Gemini est un copilote immensément puissant, mais le développeur doit agir en tant que guide méticuleux, fournissant des prompts et des exemples qui ne sont pas seulement spécifiques, mais aussi exhaustifs et exempts de biais involontaires.

3. Expériences de Génération de Code

3.1. L'Impact de la Spécificité du Prompt et du Persona (Exercice 2.1)

Cette expérience a révélé une amélioration claire et progressive de la qualité du code, directement corrélée à la qualité du prompt.

Prompt Vague : Le prompt initial ("Écris une fonction pour faire des opérations") a abouti à une fonction qui, bien que fonctionnelle, présentait un défaut de conception critique : elle retournait des messages d'erreur sous forme de chaînes de caractères. Mon analyse a conclu qu'il s'agit d'un "comportement indésirable dans un contexte professionnel", car "un code qui appelle ces fonctions pourrait s'attendre à toujours recevoir un nombre et échouer de manière imprévisible".

Prompt Spécifique : L'ajout d'exigences spécifiques a amélioré la fonction en s'assurant qu'elle était correctement nommée et gérait les arrondis, rendant sa "couverture des cas meilleure".

Prompt avec Persona : Le prompt "En tant que développeur Python..." a produit un résultat qualitativement différent. Le code est devenu "nettement plus professionnel". Le changement le plus significatif a été le passage du retour de chaînes de caractères à l'utilisation de raise ValueError pour les erreurs. C'est la pratique standard en Python car elle "sépare clairement le

flux de succès (qui retourne un nombre) du flux d'erreur (qui lève une exception), évitant ainsi les bogues silencieux".
Comme l'a conclu mon analyse :
"Le principe du persona a eu de loin le plus grand impact... La spécificité dit à l'IA quoi faire, mais le persona lui dit comment être, ce qui se traduit par une qualité de sortie bien supérieure."

3.2. L'Influence des Exemples : Prompting Few-Shot (Exercice 2.2)

Cette expérience a démontré que fournir des exemples est une technique puissante mais délicate. Mon analyse a noté que l'ajout d'exemples avait une "influence à double tranchant".
De Zero-Shot à One-Shot : Initialement, le prompt basé sur des règles (zero-shot) a produit un code parfaitement robuste. L'introduction d'un seul exemple (one-shot) a poussé l'IA à modifier sa logique de formatage pour correspondre à l'exemple, déduisant correctement que l'exemple concret (XXX-XXX-XXXX) devait prévaloir sur la règle textuelle plus ambiguë ("après le 3ème et 7ème caractère").
Le Piège du Few-Shot : Cependant, le prompt few-shot a introduit une régression. Comme noté dans mon analyse, "en fournissant un exemple d'erreur qui se concentrait uniquement sur la longueur ("SHORT"), l'IA a 'sur-appris' de cet exemple et a totalement omis la vérification des caractères alphanumériques (.isalnum())". Cela a considérablement réduit la robustesse de la fonction, la rendant vulnérable aux entrées invalides.
La principale leçon était que "la qualité vaut mieux que la quantité" en matière d'exemples. Un bon ensemble d'exemples doit être exhaustif, couvrant les cas de succès, les cas limites et, de manière cruciale, un exemple pour chaque type d'erreur attendu.

3.3. Génération d'Application : Prompts Vague vs. Détaillé (Exercice 2.3)

Le contraste dans cet exercice était frappant et a démontré l'immense valeur d'un prompt détaillé pour les tâches complexes.
Le prompt vague a produit ce que mon analyse a appelé un "prototype fonctionnel", une première ébauche nécessitant une révision importante. Visuellement, la grille CSS était "incorrectement implémentée", rompant la symétrie de la mise en page. Fonctionnellement, il n'avait "aucune protection" contre les entrées utilisateur invalides. Plus critique du point de vue de la sécurité, il

utilisait eval(), une "pratique dangereuse qui présente des failles de sécurité (XSS)".
Le prompt détaillé a agi comme un véritable "cahier des charges". Le résultat fut une "mini-application quasi-complète, robuste, sécurisée avec une expérience utilisateur réfléchie". Il a correctement implémenté le thème sombre, utilisé une grille CSS appropriée avec des espacements, mis en oeuvre une "logique de validation proactive" en JavaScript, et respecté la contrainte de sécurité en utilisant l'alternative plus sûre new Function() à eval().
Cette expérience a prouvé de manière définitive que pour les projets non triviaux, un prompt détaillé est un investissement direct dans la qualité du produit final.

4. L'IA comme Partenaire de Développement

4.1. Débogage Assisté par IA (Exercice 3.1)

Dans cette tâche, l'IA a fonctionné comme un partenaire compétent et efficace. Je lui ai fourni le code défectueux calculate_average et la trace complète de l'erreur TypeError. Comme consigné dans mon analyse, "l'IA a correctement identifié l'origine du problème". Elle a ensuite proposé une fonction améliorée qui filtrait les types de données non numériques et gérait le cas limite d'une liste vide. Le processus a également mis en évidence l'importance du prompting itératif ; l'IA a d'abord généré des tests avec unittest, et un simple prompt de suivi - "Je veux que tu réécrives ces tests en utilisant spécifiquement la syntaxe pytest" - a été nécessaire pour obtenir le résultat exact souhaité. Cela démontre que l'interaction avec une IA est un dialogue itératif.

4.2. Refactoring Dirigé par IA (Exercice 3.2)
Cet exercice a mis en évidence la puissance de l'IA lorsqu'elle est guidée par des contraintes précises. Le code initial n'était pas structuré, utilisait des noms de variables non explicites et n'était pas réutilisable. Le prompt détaillé, qui exigeait le respect de la PEP 8, des docstrings, des fonctions modulaires et un bloc if __name__ == "__main__", a transformé le script. Le tableau de mon analyse résume perfectly l'amélioration :

Le prompt avancé a guidé l'IA pour produire un code qui n'était pas seulement meilleur, mais qui

suivait les normes professionnelles du développement logiciel.

Caractéristique	Résultat du Prompt Simple	Résultat du Prompt Avancé
Structure	Une seule fonction de base.	Fonctions main et tri_par_selection séparées.
Modularité	Faible. Le code n'est pas fait pour être importé.	Élevée, grâce au bloc if __name__ == "__main__".
Documentation	Une docstring très simple.	Docstrings complètes pour le module et les fonctions.
Sécurité du code	Dangereux : Modifie la liste originale (effet de bord).	Sûr : Travaille sur une copie (.copy()), laissant l'original intact.
Clarté	Noms de variables corrects.	Noms explicites, commentaires et annotations de type (typing).
Professionnalisme	Niveau amateur.	Niveau professionnel, prêt à l'emploi.

4.3. Génération Automatisée de Documentation (Exercice 3.3)

La performance de l'IA dans cette tâche finale a été remarquable. Comme noté dans mon analyse, ma conclusion était "un grand oui", les documents générés étaient excellents. La docstring était complète et suivait une norme professionnelle, et la section README était claire et immédiatement utile.

L'aspect le plus surprenant et impressionnant a été que l'IA a amélioré le code tout en le documentant. Mon analyse le souligne :

"Elle a remplacé system_context['admins'] par system_context.get('admins', []). C'est une excellente initiative car elle rend la fonction plus sûre... L'IA n'a pas seulement documenté, elle a

aussi rendu le code plus robuste."
Cette amélioration proactive démontre le potentiel de l'IA en tant que véritable partenaire de développement capable d'améliorer la qualité du code au-delà de la demande explicite de l'utilisateur.

5. Conclusion Générale & Synthèse

Les preuves collectives de ce travail pratique établissent une conclusion indéniable : la qualité, la robustesse et le professionnalisme du code généré par l'IA sont directement et profondément proportionnels à la qualité des prompts fournis. Le parcours d'une requête vague à une spécification détaillée et basée sur un persona est un parcours d'un prototype défectueux à un actif quasi prêt pour la production.
Cette expérience a illustré de manière frappante que le rôle du développeur est en pleine évolution. L'avenir ne consiste pas simply à déléguer des tâches à une IA, mais à engager un dialogue sophistiqué avec elle. Les développeurs les plus efficaces seront ceux qui maîtriseront l'art de l'instruction - en élaborant des prompts qui servent de spécifications techniques précises et complètes. Ce passage du codeur manuel à un "architecte de l'intention" souligne l'idée que nous nous dirigeons vers un avenir de "moins de code écrit manuellement, et plus d'intelligence dans la manière de le générer". Ce travail a remodelé ma perspective, solidifiant ma conviction que l'ingénierie des prompts est une compétence fondamentale et indispensable pour le développement logiciel moderne.