Sorbonne Université
Master ANDROIDE

Système intelligent pour assister l'utilisateur à utiliser les bons outils

UE de projet M1

Nathan NYABANGANG JOKÉ TIMBA
William SARDON ARRAZ
Taha BENABOUD
Sara ELAMOURI
Elisa LI

2025

Table des matières

Chapitre 1

Introduction

Les interfaces modernes, telles que les éditeurs de texte ou de code, proposent une multitude de commandes permettant d'accélérer et d'optimiser les tâches des utilisateurs. Pourtant, nombre d'entre elles restent méconnues ou peu utilisées, même par des utilisateurs expérimentés. Ce phénomène engendre une perte de productivité et une surutilisation de commandes de base, parfois inefficaces pour certaines tâches répétitives ou complexes.

Dans ce contexte, nous avons développé une extension pour `Visual Studio Code (VS Code)` visant à assister les utilisateurs dans la découverte et l'adoption des commandes les plus efficaces pour accomplir une tâche donnée. Contrairement aux approches classiques qui suggèrent des commandes en se basant sur les séquences d'actions effectuées, notre approche repose sur l'analyse du résultat final du document pour recommander des commandes pertinentes.

Le projet s'inscrit dans le cadre de l'UE de projet du Master ANDROIDE à Sorbonne Université. Il a été réalisé par un groupe de cinq étudiants : Nathan NYABANGANG JOKÉ TIMBA, William SARDON ARRAZ, Taha BENABOUD, Sara ELAMOURI et Elisa LI. Il a été encadré par Gilles BAILLY et Julien GORI.

Le dépôt Git du projet est accessible à l'adresse suivante :
https://github.com/SAARAA136/VSCodeCommandHelper

Chapitre 2

État de l'art

Les interfaces logicielles modernes, telles que `Visual Studio Code`, proposent une grande variété de commandes permettant d'améliorer la productivité des utilisateurs. Pourtant, de nombreuses commandes restent méconnues ou peu utilisées, y compris par des utilisateurs expérimentés. C'est dans ce contexte que sont apparus des outils et méthodes visant à recommander automatiquement des commandes pour faciliter l'apprentissage et l'adoption de fonctionnalités avancées.

2.1 Outils existants

Dans le cadre de ce projet, nous avons exploré plusieurs outils et concepts liés à la recommandation de commandes. Parmi ceux-ci, `GitHub Copilot`, un assistant basé sur l'intelligence artificielle, propose des complétions de code en fonction du contexte de programmation. Copilot apprend à partir du texte tapé par l'utilisateur et tente d'anticiper les prochaines lignes de code. Nous nous sommes également intéressés à la `Command Palette` de VS Code, qui offre un accès rapide aux fonctionnalités via des raccourcis clavier ou une recherche textuelle.

Ces systèmes ont en commun d'être principalement réactifs aux actions de l'utilisateur : ils se basent sur les commandes tapées ou les séquences d'actions pour suggérer la suite à exécuter. De plus, plusieurs articles de recherche ont proposé des systèmes de recommandation utilisant l'historique des commandes pour suggérer des alternatives plus efficaces.

2.2 L'approche Dédé

Le travail le plus proche de notre approche est l'outil **Dédé**, étudié dans le cadre du projet. Dédé repose sur une idée innovante : plutôt que de suggérer des commandes à partir des actions passées, il effectue de la **postdiction** de commandes. Autrement dit, il analyse l'état courant de l'interface (par exemple, une image ou un texte) et les états précédents pour identifier une commande avancée qui aurait permis d'obtenir le même résultat avec moins d'actions.

Par exemple, si un utilisateur noircit manuellement une zone d'une image à l'aide d'un pinceau, Dédé peut reconnaître que l'état final de l'image correspond à ce qu'aurait produit la commande *« Supprimer les yeux rouges »* et proposer cette commande. Cette approche

a montré des résultats très prometteurs dans différents cas d'usage (éditeur de formes, éditeur d'images, éditeur de texte), avec des méthodes allant de règles heuristiques à des réseaux de neurones.

2.3 Prédiction vs Postdiction

La majorité des outils actuels, comme `GitHub Copilot`, `IntelliCode` ou les systèmes de raccourcis intelligents, s'appuient sur une approche de **prédiction** : ils analysent les actions passées de l'utilisateur (commandes tapées, séquences de touches, historique) pour anticiper ou recommander la prochaine commande. Cette approche a l'avantage d'être rapide, légère, et efficace lorsqu'un utilisateur suit des schémas d'action répétitifs.

Cependant, elle présente plusieurs limites :

- Elle ne fonctionne que si les actions de l'utilisateur sont bien enregistrées.
- Elle peut ignorer des intentions implicites (ex : retaper une phrase).
- Elle peut proposer des recommandations non pertinentes dans des contextes nouveaux.

À l'inverse, notre approche repose sur la **postdiction**, un concept plus récent, utilisé notamment dans l'outil Dédé. La postdiction consiste à analyser l'état final du document, sans tenir compte des actions réalisées, et à déduire quelle commande avancée aurait pu produire ce résultat.

Les bénéfices de la postdiction sont :

- Elle est indépendante de la manière dont l'utilisateur a agi.
- Elle permet de découvrir des commandes utiles même si l'utilisateur ne les connaît pas.
- Elle propose des suggestions mieux ciblées sur le contexte réel.

Ses inconvénients potentiels sont :

- Une complexité plus élevée dans la comparaison des états.
- La nécessité d'un traitement plus coûteux (machine learning, détection de patterns).
- Une dépendance à la qualité des représentations des états du document.

2.4 Notre approche

Notre extension pour `Visual Studio Code` s'inspire directement de Dédé en adoptant cette logique postdictive. Contrairement aux outils traditionnels, nous ne nous appuyons ni sur les commandes tapées, ni sur l'historique d'actions, mais uniquement sur le résultat final observé dans le document.

Ainsi, que l'utilisateur ait copié/collé un texte, utilisé un raccourci, ou simplement retapé une phrase à la main, notre outil détecte la duplication et peut proposer la commande *« Dupliquer »*. Cette approche présente un fort potentiel pour aider les utilisateurs à découvrir des commandes efficaces qu'ils n'auraient pas envisagées, et ce, sans surveiller en permanence leurs actions.

Enfin, pour construire notre prototype, nous avons mobilisé des notions étudiées en **Interaction Homme-Machine (IHM)** telles que la conception d'interfaces proactives, le retour utilisateur non-intrusif, ainsi que des premières techniques d'analyse par apprentissage automatique.

Chapitre 3

Contribution

3.1 Présentation générale de la contribution

Dans ce projet, nous avons conçu et développé *Command Helper*, une extension pour Visual Studio Code qui aide les utilisateurs à découvrir des commandes avancées susceptibles d'optimiser leur flux de travail. L'originalité de notre approche réside dans l'utilisation de la **postdiction de commandes** : plutôt que de prédire la prochaine action à effectuer à partir des actions passées, nous analysons l'état final du document pour déterminer s'il existe une commande qui aurait permis d'obtenir ce même résultat de manière plus efficace.

Cette logique postdictive s'inscrit dans la continuité des travaux réalisés sur l'outil *Dédé*, qui a montré que de telles recommandations contextuelles peuvent considérablement améliorer l'apprentissage de fonctionnalités méconnues. Nous avons adapté cette approche au contexte de l'édition de code, en tenant compte des spécificités de VS Code et en privilégiant une interface légère et non intrusive.

3.2 Concepts et méthodes étudiés

Nos travaux s'appuient sur plusieurs concepts issus de l'Interaction Homme-Machine (IHM) et de l'intelligence artificielle :

- La postdiction de commandes, qui consiste à partir de l'état final d'un document pour retrouver des séquences d'actions équivalentes, mais optimisées.
- L'analyse comportementale, via l'observation des interactions de l'utilisateur avec l'éditeur (texte, curseur, sélections).
- La détection de motifs répétitifs, permettant de proposer des raccourcis ou des alternatives plus efficaces.
- La simulation d'actions, utilisée pour tester virtuellement l'effet de certaines commandes.
- L'adaptation de l'interface, avec des recommandations intégrées de manière discrète pour ne pas perturber l'utilisateur.

Nous avons également étudié différentes représentations d'état (texte brut, curseur, sé-

lection), et envisagé leur exploitation future dans un système d'apprentissage automatique.

3.3 Évolution de l'approche technique

3.3.1 Prototype initial : système basé sur des règles

Au lancement du projet, nous avons commencé par une approche simple fondée sur des *règles prédéfinies*. Ces règles détectaient des comportements spécifiques dans l'utilisation de l'éditeur. Par exemple, si l'utilisateur sélectionnait tout le document caractère par caractère, nous pouvions lui suggérer la commande `Ctrl+A`.

Les données exploitées à ce stade étaient limitées à :

— L'état du texte dans l'éditeur,
— La position du curseur et les sélections actives.

Bien que ce système ait permis de valider le concept, il était limité à des cas très spécifiques et nécessitait une implémentation manuelle pour chaque motif à détecter.

3.3.2 Introduction de la simulation de commandes

Pour dépasser les limites des règles fixes, nous avons introduit un mécanisme de *simulation de commandes*. Cette approche consiste à appliquer, de manière virtuelle, une commande sur un état antérieur, puis à comparer le résultat simulé à l'état réel observé. Si les deux correspondent, la commande est suggérée.

Ce mécanisme a été progressivement intégré pour des commandes courantes comme :

— `Tab` (indentation),
— `Ctrl+A` (sélection du document entier),
— `Alt+↑/↓` (déplacement de lignes),
— `Ctrl+/` (commentaire de ligne).

Cette méthode, inspirée de l'approche de Dédé, a permis d'augmenter considérablement la couverture des cas détectables, tout en réduisant le besoin d'écriture de règles explicites.

3.3.3 Architecture de l'extension

Pour assurer la modularité et la maintenabilité du système, nous avons conçu une architecture structurée autour de deux composantes principales au sein même de l'extension VS Code :

— **Capture et interface (TypeScript)** :
  — Interception des événements de l'éditeur (modification du texte, mouvements du curseur),
  — Génération des états à analyser,
  — Affichage des recommandations dans un panneau latéral ou via des notifications.
— **Moteur de simulation (TypeScript)** :

— Implémentation de fonctions simulant les effets de commandes VS Code,
— Comparaison des résultats simulés avec l'état réel obtenu,
— Retour d'une commande potentiellement plus efficace.

L'ensemble du système fonctionne localement, sans dépendance à un serveur externe. Cette organisation nous a permis de faciliter les tests, la portabilité, et l'intégration directe dans l'environnement de l'utilisateur.

Afin d'assurer une séparation claire des responsabilités et de faciliter la maintenance du code, nous avons structuré le projet en trois extensions complémentaires :

— **Extension d'affichage avancé** : elle propose une interface utilisateur enrichie pour quelques commandes spécifiques, avec des éléments visuels plus détaillés (panneaux, tutoriels interactifs, etc.).
— **Extension générique** : elle permet de recommander un nombre arbitraire de commandes, à travers un affichage plus minimaliste et des interactions simplifiées. C'est cette extension qui coordonne la capture des états et la communication avec le module de simulation.
— **Extension de simulation** : elle contient les fonctions de simulation des effets des commandes, nécessaires à la comparaison des états avant et après action. Elle est invoquée par l'extension générique pour valider les suggestions.

Ce découpage modulaire permet d'expérimenter et de faire évoluer indépendamment les différents volets de l'outil, tout en gardant une cohérence fonctionnelle dans l'ensemble du système.

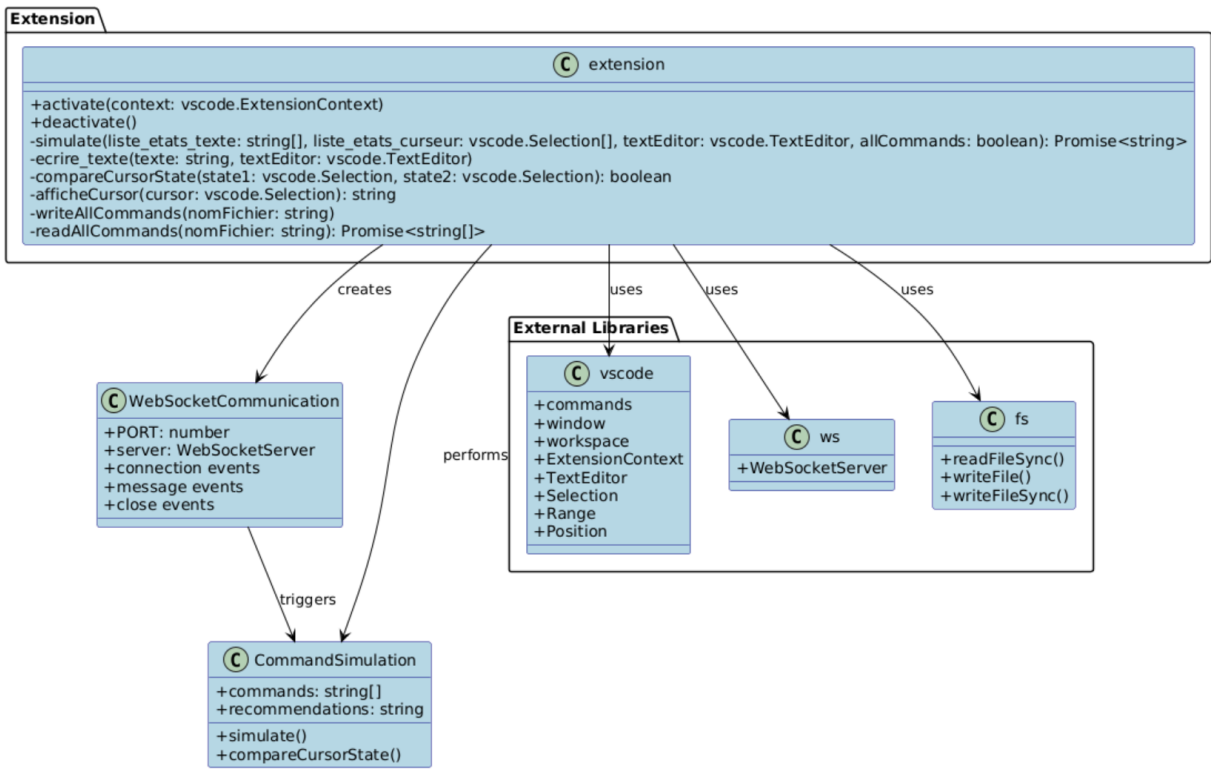


FIGURE 3.1 – Diagramme des classes

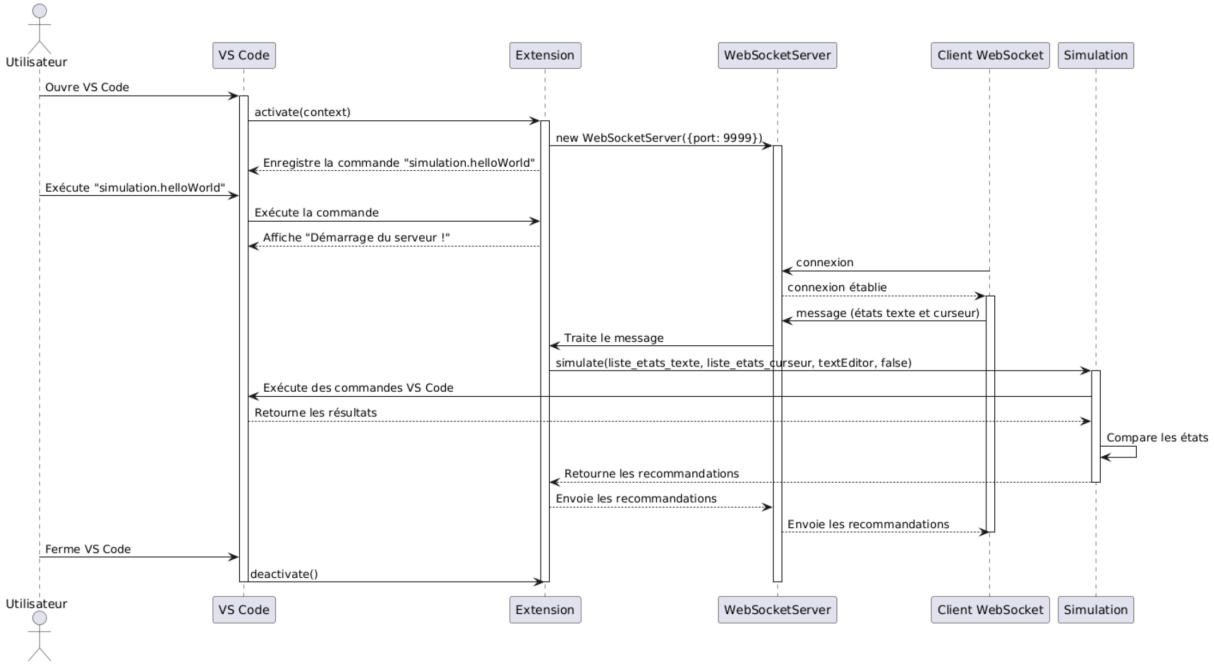


FIGURE 3.2 – Diagramme de séquence

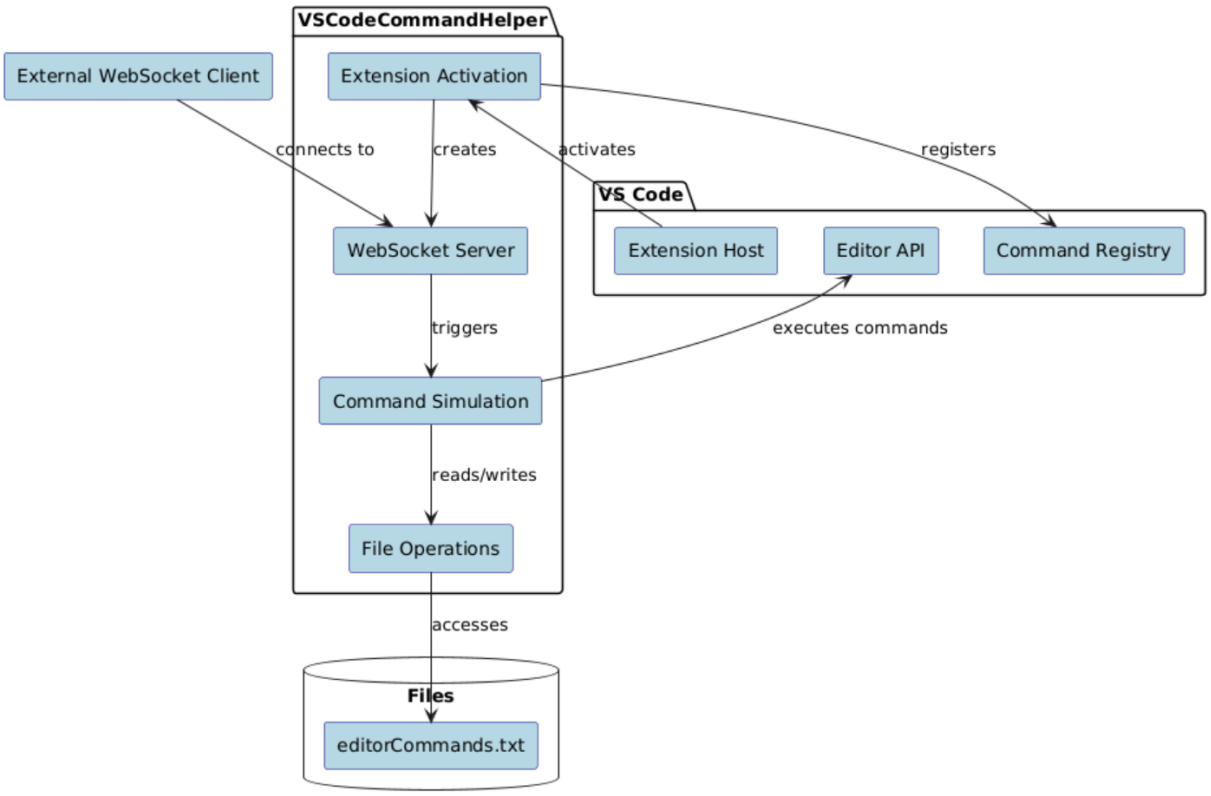


FIGURE 3.3 – Diagramme de composants

3.3.4 Enrichissements progressifs

Au fil du développement, plusieurs fonctionnalités ont été ajoutées :

— Interface graphique avec panneau latéral interactif et notifications,
— Mécanisme de mémoire des préférences utilisateur (recommandations ignorées ou acceptées),
— Ajout d'un historique de recommandations.

3.4 Résultats obtenus

3.4.1 Fonctionnement du système

Nous avons obtenu un système fonctionnel capable de :

— Capturer des séquences d'états en temps réel,
— Simuler différentes commandes sur des états antérieurs,
— Recommander une commande lorsque l'état simulé correspond à l'état observé.

Les cas suivants ont été testés avec succès :

— Sélection manuelle vs `Ctrl+A`,
— Duplication de ligne vs `Ctrl+D`,
— Déplacement de ligne par copier-coller vs `Alt+↑/↓`,
— Commentaire manuel vs `Ctrl+/`.

3.4.2 Retour sur l'expérience utilisateur

Les premiers retours ont montré que le système de recommandations est :

— **Non-intrusif** : la notification est visible mais ne bloque pas l'utilisateur,
— **Compréhensible** : le panneau permet d'expliquer clairement la commande proposée,
— **Pertinent** : les recommandations se déclenchent uniquement lorsqu'un gain d'efficacité est détecté.

De plus, les utilisateurs peuvent personnaliser l'expérience, en refusant certaines suggestions ou en désactivant celles jugées inutiles.

3.5 Analyse du temps d'exécution

Afin de mieux comprendre l'impact de la taille des données sur la performance du système de recommandation, nous avons mesuré le temps d'exécution d'une simulation de postdiction en fonction du nombre d'états testés (N), dans deux scénarios :

— un cas avec une seule commande simulée ($C = 1$),
— un cas avec dix commandes simulées ($C = 10$).

Pseudo-code analysé

```
Pour i allant de 1 à N faire :
    exécuter(C)
    enregistreRecommandation()
FinPour
```

Chaque état du document est comparé à l'effet simulé de C commandes. On suppose que chaque simulation de commande prend un temps constant T.

Analyse de complexité

Le temps d'exécution total est donc proportionnel à $N \times C \times T$, soit une complexité en :

$$\mathcal{O}(N \cdot C)$$

Résultats expérimentaux

Les graphiques ci-dessous présentent l'évolution empirique du temps d'exécution mesuré en fonction du nombre d'états, pour $C = 1$ puis $C = 10$.

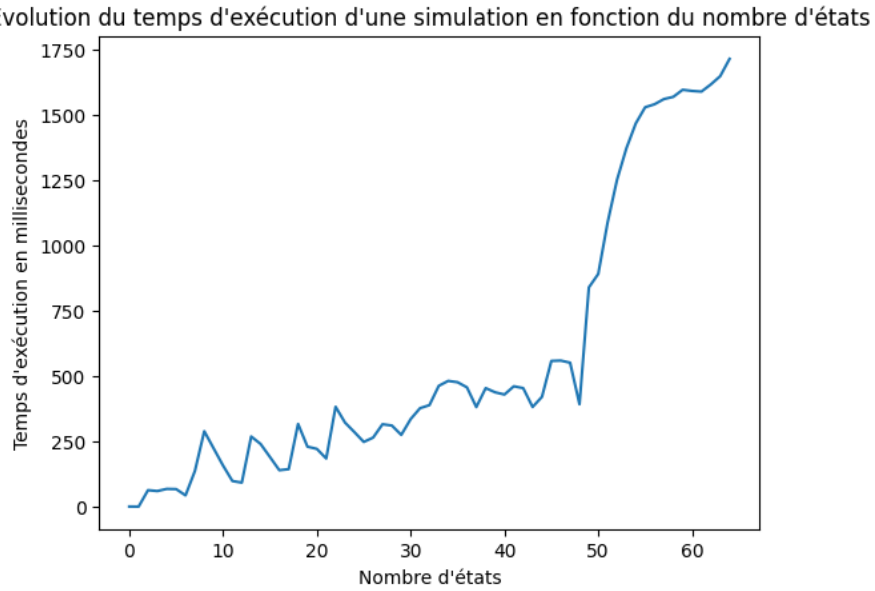


FIGURE 3.4 – Temps d'exécution en fonction du nombre d'états ($C = 1$)

La courbe présente des fluctuations locales entre 0 et 50 états, avec un temps compris entre 0 et 500 ms. Au-delà de 50 états, on observe une augmentation brutale jusqu'à environ 1750 ms.

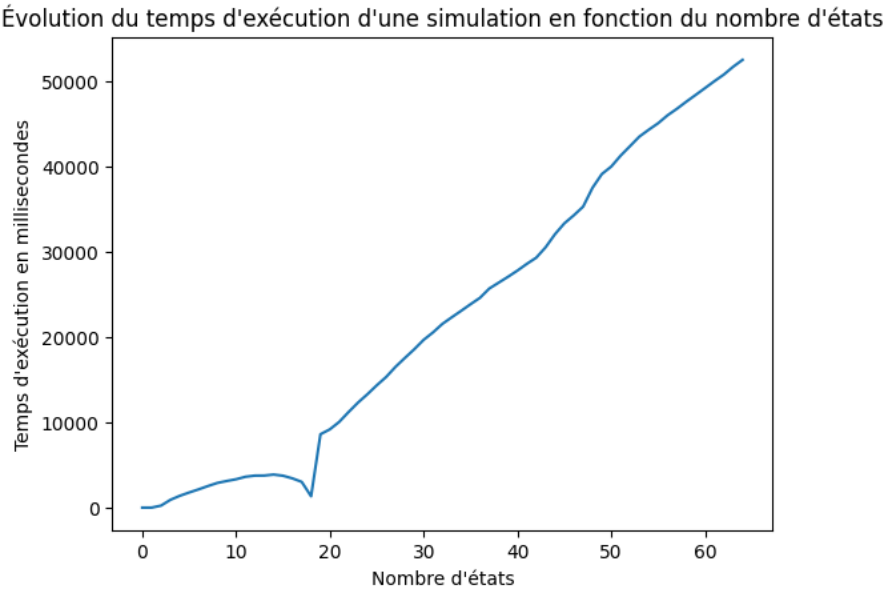


FIGURE 3.5 – Temps d'exécution en fonction du nombre d'états ($C = 10$)

Le temps d'exécution suit une tendance globalement linéaire, mais présente une rupture aux alentours de 17 états, avant de reprendre une croissance régulière.

Les figures **Figure 3.1** ($C = 1$) et **Figure 3.2** ($C = 10$) illustrent deux comportements distincts du temps d'exécution en fonction du nombre d'états simulés.

Dans la **Figure 3.1**, on observe des fluctuations locales entre 0 et 50 états, avec un temps compris entre 0 et 500 ms. À partir du 50^{e} état, la courbe montre une augmentation marquée, atteignant environ 1750 ms. Ce comportement suggère l'activation tardive d'un processus coûteux ou une accumulation d'opérations.

La **Figure 3.2**, quant à elle, suit une tendance globalement linéaire, mais présente une rupture visible aux alentours de 17 états. Après cette inflexion, la croissance reprend de manière régulière. Ce type de discontinuité peut signaler un changement de stratégie interne ou un ajustement structurel du système.

Ces résultats confirment la complexité théorique en $\mathcal{O}(N \cdot C)$, tout en mettant en évidence des phénomènes non linéaires ponctuels qui mériteraient une analyse complémentaire pour améliorer la stabilité et les performances du système.

3.6 Discussion et comparaison avec l'état de l'art

Notre approche se distingue des systèmes prédictifs traditionnels comme GitHub Copilot ou IntelliCode, qui se concentrent principalement sur la complétion de code en temps réel. Elle se différencie également des extensions statiques comme Keybinding Analyzer, qui n'offrent aucune interaction contextuelle.

En adoptant une logique postdictive, *Command Helper* propose une assistance ciblée, déclenchée uniquement lorsque l'état du document suggère qu'une commande aurait pu être plus appropriée. Cela permet non seulement de respecter le flux de travail de l'utilisateur,

mais aussi de l'amener progressivement à découvrir et intégrer de nouvelles pratiques plus efficaces.

Cette approche présente toutefois certaines limites :

— La couverture des commandes reste partielle et nécessite un travail manuel pour chaque simulation,
— Les performances peuvent être impactées sur de gros fichiers ou lors d'un enchaînement rapide d'actions,
— Le système n'intègre pas encore la notion de profils utilisateurs ou d'adaptation à long terme.

3.6.1 Discussion autour de l'approche par simulation

L'approche par simulation s'avère impraticable en pratique, car elle implique l'exécution de l'ensemble des commandes pour tous les états possibles à chaque création d'un nouvel état. Or, ces créations surviennent chaque fois que l'utilisateur déplace le curseur ou modifie un caractère dans le texte. Ce processus devient rapidement trop coûteux en temps dès lors que le nombre d'états devient important. De plus, la duplication complète du document de travail à chaque exécution de commande engendre une surcharge considérable, particulièrement lorsque le fichier contient un grand volume de texte.

Un autre obstacle réside dans le manque de précision des recommandations de commandes. En effet, l'API de VS Code n'étant pas conçue pour la simulation, certaines commandes déclenchent l'ouverture de nouvelles fenêtres ou redirigent l'utilisateur vers la définition de fonctions, ce qui interrompt prématurément le programme. Cette limitation réduit considérablement l'intérêt d'une approche générique, puisqu'il devient nécessaire de vérifier manuellement chaque commande au préalable afin de s'assurer qu'elle n'entraîne pas d'interruption. Ce processus revient, en pratique, à un ajout manuel de chaque commande, ce que l'on cherchait justement à éviter.

Nous avons donc opté pour une approche restreinte, en limitant la simulation à dix commandes sélectionnées à l'avance. Idéalement, l'API de VS Code devrait proposer un mécanisme de filtrage permettant d'exécuter uniquement les commandes compatibles avec un environnement de simulation, sans risque de blocage.

Chapitre 4

Conception de l'Interface Homme-Machine (IHM)

L'un des objectifs centraux du projet *Command Helper* était de fournir une assistance efficace tout en minimisant la perturbation du flux de travail de l'utilisateur. Cela relève directement des principes de l'Interaction Homme-Machine (IHM), et plusieurs choix de conception ont été réalisés en ce sens.

4.1 Objectifs d'interaction

Notre extension vise trois objectifs principaux :

- **Discrétion** : ne pas interrompre l'utilisateur dans son travail.
- **Clarté** : rendre immédiatement compréhensible la suggestion proposée.
- **Apprentissage progressif** : permettre à l'utilisateur d'explorer de nouvelles commandes sans surcharge cognitive.

4.2 Composants de l'IHM développée

4.2.1 Popup de recommandation

Lorsqu'une commande pertinente est détectée, une petite boîte de notification s'affiche en bas à gauche de l'éditeur. Celle-ci indique le raccourci clavier ou la commande correspondante. L'utilisateur peut :

- cliquer pour en savoir plus ;
- ignorer la recommandation ;
- désactiver les futures suggestions pour cette commande.

4.2.2 Panneau latéral de tutoriel

Un panneau latéral explicatif peut être déclenché, contenant :

- une explication textuelle de la commande ;
- une séquence d'images montrant comment elle fonctionne ;
- (optionnellement) une courte vidéo démonstrative.

4.2.3 Interface personnalisable

L'utilisateur peut accéder à un menu de préférences pour :

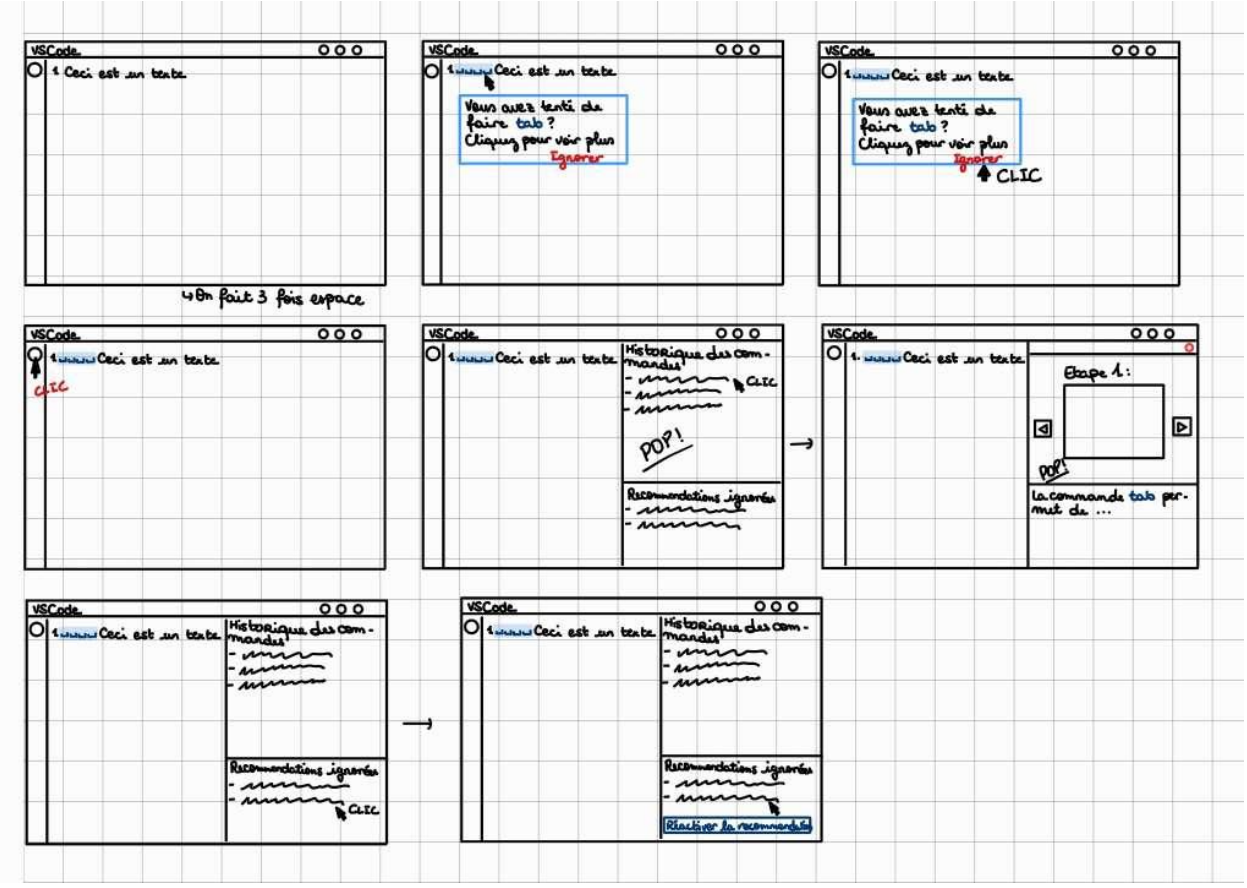
— ajuster la fréquence des suggestions ;
— consulter l'historique des recommandations ;
— réactiver ou ignorer certaines commandes.

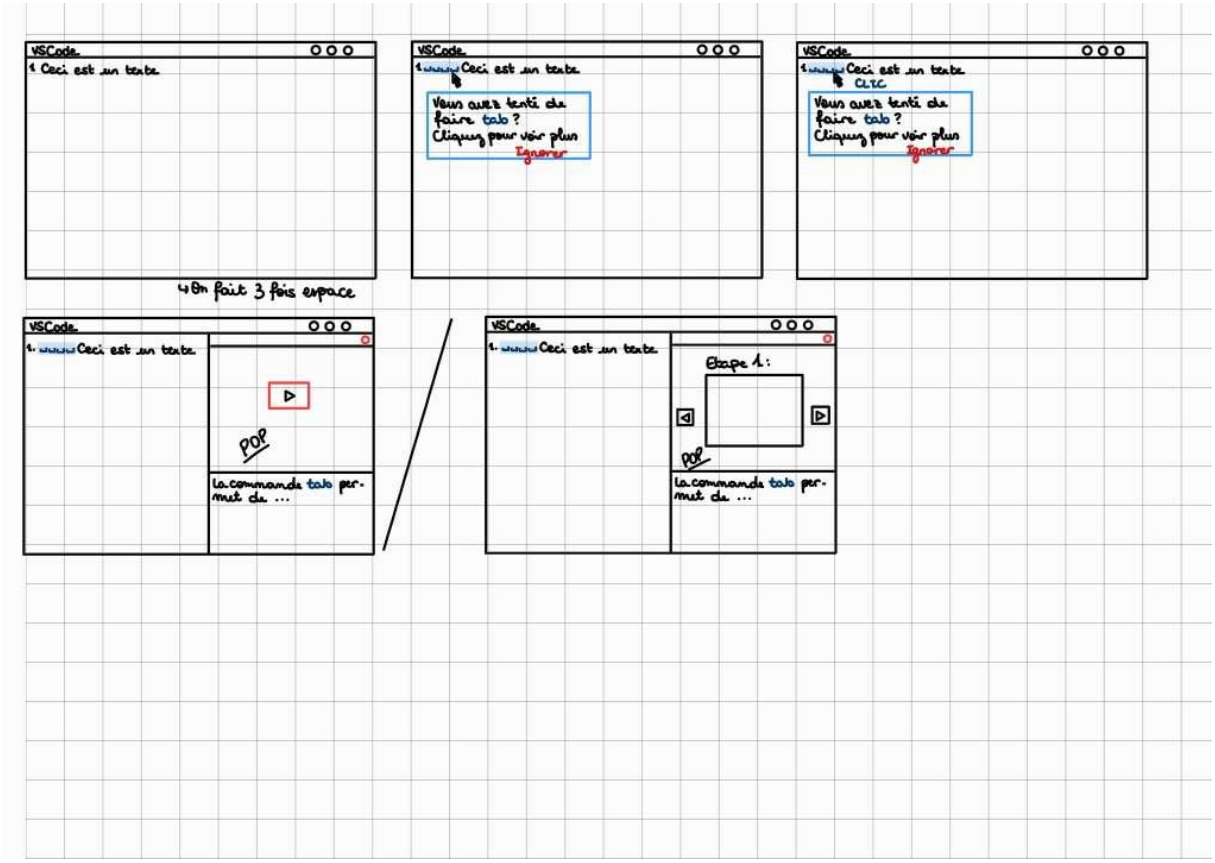
4.3 Choix de design et principes IHM mobilisés

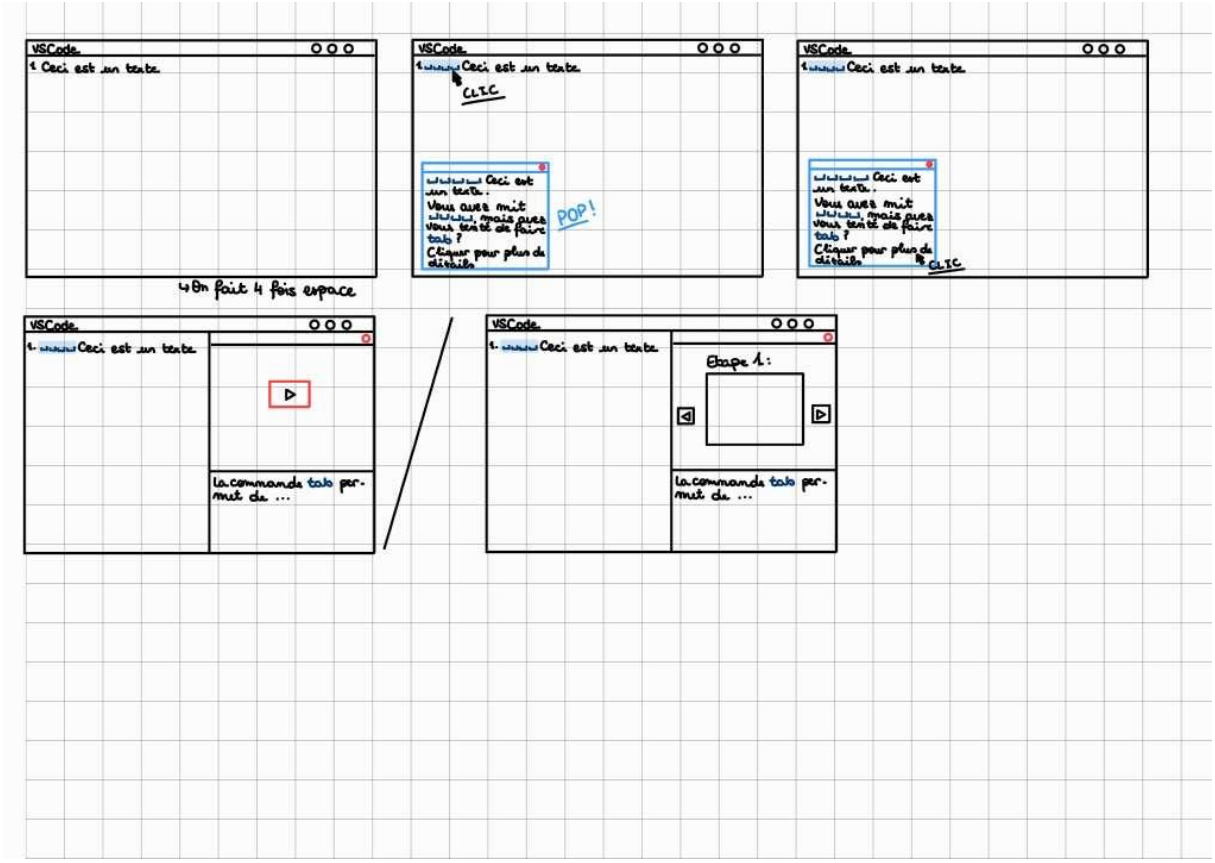
— **Retour utilisateur immédiat** : l'utilisateur voit les suggestions au moment où elles deviennent pertinentes.
— **Non-intrusion** : le système ne bloque jamais l'édition du code.
— **Proactivité** : les suggestions anticipent les besoins sans perturber.
— **Tolérance à l'erreur** : l'utilisateur peut ignorer, désactiver ou réactiver les recommandations.

Ces éléments s'appuient sur les heuristiques de Nielsen et les principes de conception centrée utilisateur.

4.4 Schéma global d'interaction




VSCode
1 Ceci est un texte.
Vous avez tenté de faire tab ?
Cliquez pour voir plus
Ignorer
CLIC
↳ On fait 3 fois espace
POP
la commande tab permet de ...
Etape 1 :


VSCode
1 Ceci est un texte.
CLIC
Ceci est un texte.
Vous avez mit plus que vous teniez de faire tab ?
Cliquez pour plus de détails
POP !
↳ On fait 4 fois espace
la commande tab permet de ...
Etape 1 :

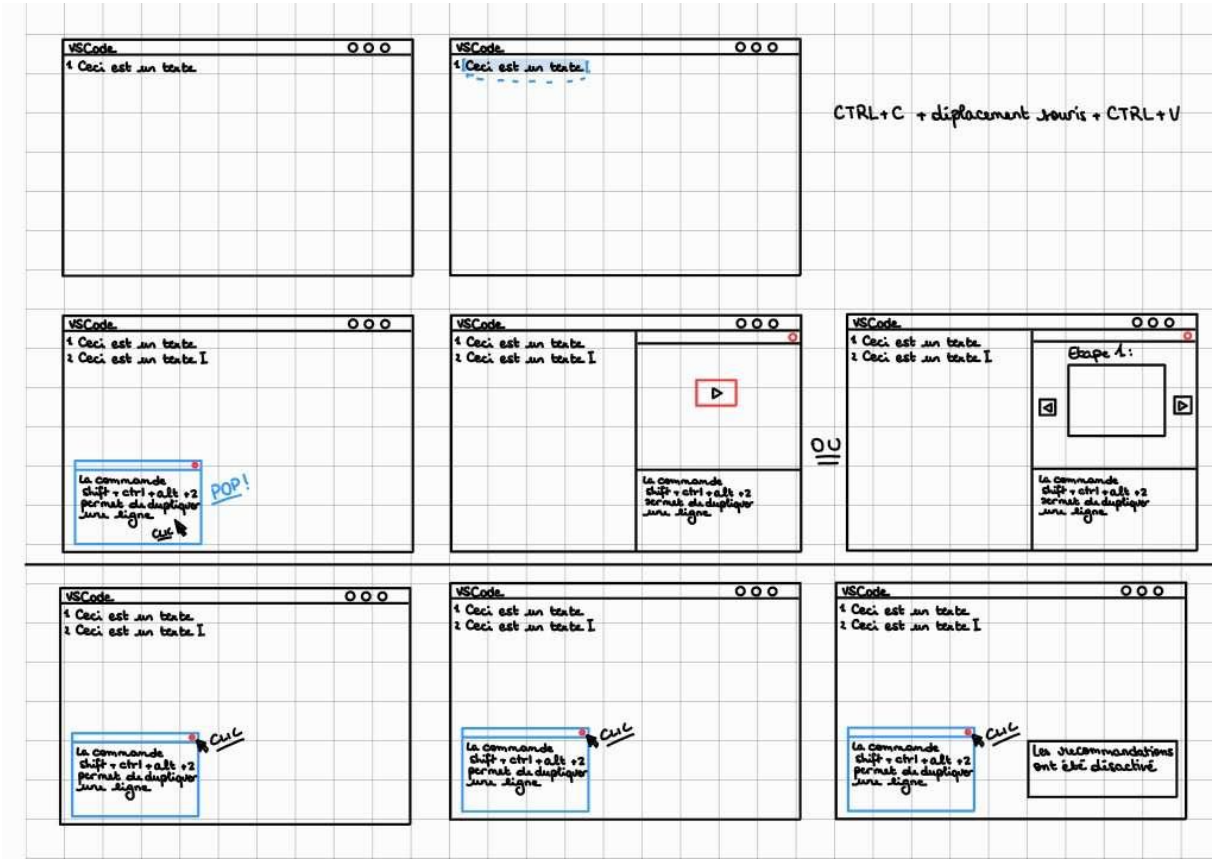


FIGURE 4.1 – Schéma des différentes étapes interactives de l'extension *Command Helper*.

Ce schéma présente les étapes clefs de fonctionnement : de l'observation des états, à la simulation de commande, en passant par la détection d'un équivalent et l'affichage contextuel.

4.5 Évolutions possibles de l'IHM

Parmi les pistes envisagées :

- **Mode "ne pas déranger"** : Permettre à l'utilisateur de désactiver temporairement les recommandations en un clic, via une icône ou un raccourci clavier, lorsqu'il souhaite se concentrer.
- **Mode tutoriel guidé** : Proposer un tutoriel interactif lors de la première apparition d'une commande, avec étapes visuelles intégrées (comme un pas-à-pas en surimpression ou dans une pop-up latérale).
- **Résumé de productivité périodique** : Générer un tableau de bord hebdomadaire ou mensuel résumant les gains potentiels apportés par les recommandations acceptées, pour encourager leur adoption continue.

En conclusion, le soin apporté à l'IHM contribue à faire de *Command Helper* un assistant réellement utilisable, qui guide sans imposer, et respecte les principes fondamentaux de l'expérience utilisateur dans les environnements de développement professionnels.

Chapitre 5

Conclusion

Dans le cadre de ce projet, nous avons conçu et développé *Command Helper*, une extension pour Visual Studio Code dont l'objectif principal était d'aider les utilisateurs à découvrir et à adopter les commandes les plus efficaces, non pas à partir des séquences d'actions effectuées, mais par l'analyse de l'état final du document (postdiction).

Les résultats obtenus confirment la pertinence de notre approche. En effet, nous avons pu démontrer expérimentalement qu'il était possible de détecter des optimisations dans les actions utilisateurs avec une complexité linéaire ($\mathcal{O}(N \cdot C)$), même si des phénomènes ponctuels de rupture apparaissent pour certains seuils de nombre d'états analysés.

Les retours utilisateurs ont souligné trois points forts majeurs :

- Une interface intuitive et non intrusive, permettant une intégration fluide dans les habitudes de travail.
- Des recommandations pertinentes, apportant un réel gain d'efficacité.
- Une architecture modulaire facilitant à la fois l'évolution et la maintenance du système.

Cependant, notre travail présente également des limites identifiées :

- La nécessité d'implémenter manuellement la simulation pour chaque commande spécifique limite la généralisation rapide à un grand nombre de commandes.
- Des performances pouvant être impactées ponctuellement lors de la gestion d'un grand nombre d'états ou lors d'enchaînements rapides d'actions.
- L'absence actuelle d'un mécanisme adaptatif sur le long terme, capable de personnaliser davantage les recommandations selon les profils utilisateurs.

Ces points soulignent les perspectives de poursuite que nous envisageons :

- Développer un moteur de postdiction reposant sur l'apprentissage automatique, afin d'étendre automatiquement la couverture à davantage de commandes.
- Mettre en place une stratégie dynamique de gestion des ressources pour optimiser les performances en conditions réelles, notamment en ajustant le nombre de commandes simulées.
- Intégrer un système de profils utilisateurs, afin d'offrir une expérience véritablement personnalisée et évolutive.
- Réaliser une évaluation à plus grande échelle et sur une durée prolongée afin d'étudier précisément les effets sur l'apprentissage et l'adoption des commandes recommandées.

En conclusion, *Command Helper* ouvre des perspectives innovantes pour l'amélioration de la productivité dans les environnements de développement. Ce projet démontre la viabilité et la pertinence de la postdiction appliquée aux éditeurs de texte, et pose les fondations d'un système intelligent capable d'accompagner durablement les utilisateurs vers une utilisation optimale de leurs outils quotidiens.

Chapitre 6

Bibliographie

[appert2009strokes]
[cockburn2014novice]
[do2023err]
[giannisakis2017iconhk]
[grossman2009survey]
[malacria2013exposehk]
[matejka2009applications]
[matejka2013patina]
[murphy2012environment]
[murphy2019toilet]
[nielsen1994usability]
[ofer2021language]
[1]
[2]
[sears1994splitmenus]
[twidale2005shoulder]

Annexe A

Cahier des charges

Présentation du Projet

Contexte

Les interfaces logicielles proposent une multitude de commandes, mais nombre d'entre elles restent méconnues ou sous-utilisées par les utilisateurs.

Par exemple, dans un éditeur d'image, un utilisateur pourrait peindre manuellement en noir des yeux rouges plutôt que d'utiliser la commande spécifique "Enlever les yeux rouges". Dans un éditeur de texte, il pourrait recopier une phrase au lieu d'utiliser "Copier/Coller" ou la commande plus efficace "Dupliquer". L'objectif est donc de proposer un assistant intelligent capable de suggérer les commandes les plus adaptées en fonction du résultat obtenu dans le document, et non uniquement sur la base des actions effectuées.

Objectif

Développer une extension pour VS Code qui aide les utilisateurs à identifier et adopter les commandes les plus efficaces dans leur workflow. Cet assistant intelligent analysera les modifications apportées au document et suggérera des commandes optimisées pour accomplir des tâches récurrentes plus rapidement et efficacement.

Approche

L'approche proposée repose sur trois étapes principales :

— Apprentissage des effets des commandes : Utiliser des techniques de Machine Learning pour analyser comment chaque commande impacte un document texte.
— Détection des résultats obtenus : Observer l'état du document et identifier les modifications effectuées (ex. une phrase dupliquée, une suppression répétée).
— Suggestion de commandes optimisées : Proposer à l'utilisateur la ou les commandes les plus efficaces pour produire le même résultat.

Originalité de l'approche

Contrairement aux techniques actuelles qui suggèrent des commandes en analysant la séquence d'actions effectuées par l'utilisateur (ex. proposer "Dupliquer" au lieu de "Copier" + "Coller"), cette approche repose sur l'analyse du résultat final indépendamment des actions utilisées.

Ainsi, que l'utilisateur ait dupliqué du texte via :

— Copier/Coller via raccourci clavier
— Copier/Coller via la souris
— Réécriture manuelle du texte

L'assistant sera capable de reconnaître l'intention et de recommander la commande la plus pertinente pour optimiser l'action.

A.1 Besoins et exigences

L'extension doit permettre aux utilisateurs de découvrir et d'adopter des commandes optimales dans leur flux de travail, en réduisant les manipulations redondantes et inefficaces. Elle doit être intuitive et non intrusive, en suggérant des alternatives pertinentes sans interrompre le travail de l'utilisateur. De plus, l'extension doit être compatible avec les différentes configurations et systèmes d'exploitation supportés par VS Code. Un système de suivi des interactions doit être mis en place afin d'améliorer continuellement les suggestions en fonction des habitudes des utilisateurs.

L'extension doit être développée sous forme d'un plugin VS Code et intégrée avec l'API de l'éditeur pour analyser les modifications effectuées dans la fenêtre d'édition. Elle devra détecter des motifs spécifiques d'actions effectuées par l'utilisateur (comme un copier-coller manuel) et proposer une commande alternative plus efficace via une notification contextuelle. L'utilisateur doit pouvoir interagir avec cette notification pour obtenir plus d'informations sous forme d'aide interactive (vidéo, images, explications textuelles). Une option pour désactiver certaines suggestions ou ajuster la fréquence d'apparition des notifications doit être intégrée. Enfin, l'extension doit inclure un système de journalisation des actions afin de permettre une analyse et une amélioration continue du modèle de prédiction, tout en respectant les principes de confidentialité et de performance.

Le but sera de créer un classifieur qui à chaque état, simule toutes les commandes avancées séparément puis compare avec le prochain état émis par l'utilisateur pour prédire les commandes à recommander.

Tester et démontrer l'efficacité de la solution d'un point de vue IHM.

A.2 Solution envisagée

L'extension VS Code proposera à l'utilisateur les fonctionnalités suivantes :

Commande	Description de la commande
TAB	Permet de faire une tabulation
Ctrl+A	Permet de sélectionne tout le texte du fichier
Ctrl+Shift+ :	Permet de commenter la ligne ou le bloc de texte
Alt+↑/↓	Permet de déplacer la ligne ou le bloc de texte vers le haut/bas
Ctrl+Shift+L	Permet de sélectionner toutes les occurrences du mot sélectionné
Ctrl+Z	Permet d'annuler la dernière action réalisée
Ctrl+Y	Permet d'annuler l'annulation de la dernière action réalisée
Ctrl+D	Permet de sélectionner la prochaine occurrence du mot sélectionné
Ctrl+Shift+K	Permet de supprimer la ligne courante

TABLE A.1 – Liste des commandes à implémenter

— L'utilisateur entre une série de touches et si l'extension repère une commande qui correspond à cette série, propose sous forme de pop-up qui apparaît en bas à gauche de la fenêtre cette commande ;
— Autre possibilité : le résultat d'une commande est souligné et l'utilisateur peut passer la souris sur ce soulignage pour ouvrir une fenêtre latérale correspondant à la fenêtre "tutoriel" de l'extension ;
— La fenêtre "tutoriel" représente la manière dont l'utilisateur peut utiliser la commande recommandée sous forme d'illustrations parmi lesquelles l'utilisateur peut naviguer avec les flèches.
— Si l'utilisateur ferme 3 fois la fenêtre pour une même commande, l'extension arrête de proposer cette commande ou alors
— Si l'utilisateur clique sur cette pop-up, cela ouvre une fenêtre d'aide qui montre ce que fait la commande ;
— La fenêtre d'aide peut montrer ce que fait la commande sous forme de vidéo, d'images étape par étape dans lequel on se déplacer à l'aide de flèches ;
— Enregistre ce que fait l'utilisateur sous forme de fenêtre d'historique ;
— Possibilité éventuelle de changer de modèle de machine learning ;
— Historique de toutes les recommandations précédentes de l'utilisateur ;
— Liste des recommandations ignorées que l'utilisateur pourra personnaliser à sa guise (demander à afficher ou ne plus afficher les extensions sous forme de cases à cocher) ;
— Adapte les commandes proposées en fonction de l'OS de l'utilisateur ;

A.3 Scénarios d'utilisation

Scénario 1 : Étudiante (Emma)

Emma, une étudiante en première année d'informatique, installe VS Code et l'extension sur sa machine. Elle ouvre un nouveau fichier de code et commence à travailler. En tapant du code, elle réalise qu'elle répète souvent des actions manuelles, comme commenter des blocs de code ou déplacer des lignes.

Pendant qu'elle sélectionne plusieurs lignes pour les commenter, un message s'affiche en bas à gauche de l'écran : **Utilisez Ctrl+Shift+/ pour commenter rapidement ce**

bloc. Emma essaie la commande et constate que c'est plus rapide que de commenter manuellement.

Plus tard, alors qu'elle réorganise son code, un autre message lui propose : **Déplacez cette ligne rapidement avec Alt+↑ ou Alt+↓.** Emma utilise la commande et améliore son efficacité.

Grâce à ces suggestions, Emma découvre des commandes utiles et optimise son workflow.

Scénario 2 : Professionnel (Lucas)

Lucas, un développeur professionnel, utilise VS Code pour travailler sur un projet avec de nombreux fichiers. Il a l'habitude de coder rapidement, mais il ne connaît pas toutes les commandes avancées de VS Code.

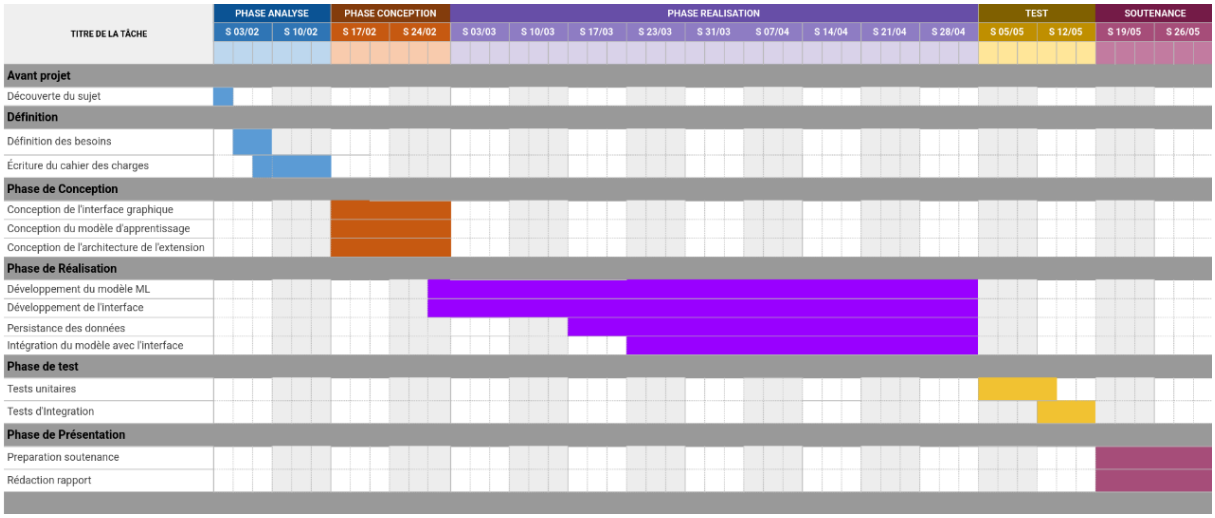
Alors qu'il modifie un mot à plusieurs endroits dans son code, un message s'affiche en bas à gauche de l'écran : **Appuyez sur Ctrl+Shift+L pour modifier ce mot partout en même temps.** Lucas essaie la commande et constate que cela accélère son travail.

Plus tard, alors qu'il navigue dans un fichier volumineux, un autre message lui propose : **Utilisez Ctrl+D pour sélectionner la prochaine occurrence du mot sélectionné.** Lucas adopte la commande et gagne du temps en évitant de rechercher manuellement les occurrences.

Grâce à ces suggestions, Lucas découvre des commandes avancées qui améliorent sa productivité.

A.4 Planification et livrables

A.4.1 Diagramme de Gantt prévisionnel du projet



Annexe B

Manuel utilisateur

B.1 Présentation de l'extension VSCodeCommandHelper

VSCodeCommandHelper est une extension pour `Visual Studio Code` qui assiste les développeurs en suggérant les commandes les plus optimales selon le contexte et les actions effectuées. Ce projet combine intelligence artificielle, développement logiciel et interaction homme-machine (IHM) pour améliorer l'expérience utilisateur et optimiser l'usage des fonctionnalités de l'éditeur.

Ce projet est encadré par **Gilles Bailly** et **Julien Gori**.

B.2 Prérequis et installation

Pour utiliser l'extension, vous devez remplir les prérequis suivants :

— Un terminal `bash` avec les droits d'exécution,
— `Node.js` installé,
— Une version récente de `Visual Studio Code`.

Note pour utilisateurs Mac : Veillez à avoir la commande `code` disponible dans votre terminal (via la palette de commande : *Shell Command : Install 'code' command in PATH*).

Le projet se compose de trois extensions :

1. Une extension avec un affichage **avancé** pour quelques commandes basiques,
2. Une extension plus **générique** avec un affichage simple, capable de recommander un nombre arbitraire de commandes,
3. Une extension dédiée à la **simulation** des commandes, nécessaire au fonctionnement de l'extension générique.

Installation

Depuis le répertoire racine `VSCodeCommandHelper`, exécutez la commande suivante :

```
./install
```

B.3 Utilisation des extensions

B.3.1 Extension à affichage avancé

Depuis le dossier `VSCodeCommandHelper`, lancez :

```
code commandhelper-rules
```

Une fenêtre `VS Code` s'ouvre. Ouvrez ensuite le fichier `extension.ts`, puis appuyez sur **F5** pour activer l'extension. Une nouvelle fenêtre s'ouvre, dans laquelle l'utilisateur peut écrire du texte ou du code : le système proposera alors certaines commandes potentiellement utiles.

*Pour redémarrer une simulation, relancez **F5** depuis le fichier extension.ts.*

B.3.2 Extension générique par simulation

L'approche par simulation consiste à :

- Créer un nouvel état à chaque modification (déplacement du curseur ou modification du texte),
- Reproduire cet état dans une fenêtre dédiée (simulation),
- Exécuter les commandes à tester et comparer leur effet avec l'état de la fenêtre de travail (commandhelper),
- Recommander la commande si les états correspondent.

Lancement :

Depuis le dossier `VSCodeCommandHelper`, exécutez :

```
./start
```

Cela ouvre deux fenêtres :

- Une fenêtre *simulation* pour rejouer les commandes à tester,
- Une fenêtre *commandhelper* dans laquelle l'utilisateur travaille.

Étapes :

1. Dans la fenêtre *simulation*, ouvrez le fichier `extension.ts` et appuyez sur **F5** pour lancer la fenêtre de débogage. Si une sélection vous est proposée, choisissez « *VS Code Extension Development*».
2. Dans la fenêtre *commandhelper*, ouvrez également `extension.ts`, puis appuyez sur **F5**. La nouvelle fenêtre qui s'ouvre sera l'environnement de travail de l'utilisateur, dans lequel il pourra écrire librement.

Lorsque le système identifie une commande pertinente, une fenêtre pop-up s'affiche avec :

- Le nom de la commande,

— Un bouton « *Voir* ».

En cliquant sur « *Voir* », un panneau latéral s'ouvre avec la **liste des commandes recommandées** (cases à cocher).

— Décochez une commande pour **cesser sa recommandation**,
— Recochez-la pour **la réactiver**.

Note : Plus l'utilisateur écrit, plus la simulation peut devenir longue (voir le rapport pour plus de détails).

Redémarrage de la simulation :

Pour redémarrer la simulation, répétez la séquence suivante :

1. **F5** dans le fichier `extension.ts` de *simulation*,
2. Puis **F5** dans le fichier `extension.ts` de *commandhelper*.

B.4 Illustrations de l'interface

Les schémas fournis dans ce rapport (voir figures associées) présentent les différentes interfaces et interactions envisagées dans les premières phases de conception de l'extension. Ils illustrent comment l'utilisateur interagit avec les notifications, les fenêtres contextuelles et les panneaux latéraux pour une expérience utilisateur optimisée.