Coder avec l'IA Générative

Créer un modèle de données d'une solution I.A en utilisant des méthodes de Data science

Objectifs de l'Atelier

Maîtriser les Outils

Apprendre à utiliser efficacement GitHub Copilot, TabNine et IntelliCode pour accélérer le développement et améliorer la qualité du code.

Personnaliser l'IA

Adapter les assistants de code aux besoins spécifiques de votre équipe grâce aux instructions personnalisées et aux configurations avancées.

Comprendre l'Impact

Évaluer l'influence de l'IA générative sur la productivité, la qualité du code et les pratiques de développement modernes.

Pourquoi Utiliser des Assistants de Code ?

Les assistants de code s'appuient sur des modèles de langage entraînés sur de vastes corpus de code source. Ces outils révolutionnent le développement logiciel en offrant des capacités avancées qui transforment la manière dont nous écrivons du code.

Capacités Principales

- **Autocomplétion contextuelle** : suggestions complètes basées sur le contexte du projet
- **Génération de code** : création de fonctions entières à partir de descriptions en langage naturel
- **Détection d'erreurs** : identification proactive de bugs et suggestions de corrections
- **Traduction de code** : conversion entre différents langages de programmation
- **Documentation automatique** : génération de commentaires et résumés de fonctions

Selon les rapports de 2025, ces outils reposent sur des modèles comme OpenAI Codex, Claude ou des modèles propriétaires capables d'analyser le contexte et de proposer des suggestions cohérentes.

Impact Mesurable

GitHub Copilot a démontré une amélioration significative de la productivité et de la satisfaction des développeurs dans plusieurs études indépendantes.

Important : L'IA générative n'est pas un remplacement des développeurs, mais un outil d'assistance. Toute suggestion doit être relue, testée et validée manuellement avant intégration.

Responsabilité et Limites de l'IA

L'utilisation responsable des assistants de code nécessite une compréhension claire de leurs capacités et limitations. Les développeurs doivent adopter une approche critique et vigilante.

Vérification Systématique

L'IA peut générer du code erroné, inefficace ou vulnérable. Chaque suggestion doit être examinée attentivement, testée dans différents scénarios et adaptée aux besoins spécifiques du projet. Ne jamais accepter aveuglément une suggestion.

Sécurité Avant Tout

Évitez d'inclure des mots de passe, clés API ou données sensibles en clair dans le code. Protégez les ressources critiques et suivez les bonnes pratiques de sécurité établies par votre organisation.

Limitations Techniques

Les modèles peuvent être biaisés par leurs données d'entraînement et moins performants dans des langages peu représentés. La portée des suggestions est limitée au contexte fourni.

Conformité Légale

Certaines industries ou projets imposent des restrictions strictes sur l'utilisation de l'IA. Vérifiez les obligations légales et réglementaires applicables à votre contexte professionnel.

GitHub Copilot : L'Assistant Intelligent

Fonctionnalités Principales

Assistant de codage intégré : Copilot propose du code au fil de la frappe pour plusieurs langages et frameworks. Il génère des lignes ou des fonctions entières et explique des concepts directement dans l'éditeur.

Disponibilité : Visual Studio Code, Visual Studio, Neovim, JetBrains et sur le site GitHub.com.

Copilot Chat : Interface conversationnelle permettant de poser des questions sur le code, demander des explications ou des corrections. Les agents Copilot peuvent même créer des pull requests et effectuer des changements autonomes (plan payant).

Personnalisation Avancée

Les **instructions personnalisées** permettent de préciser des règles de style, des frameworks utilisés ou des conventions de codage. Elles peuvent être :

- **Personnelles** : définies par l'utilisateur pour toutes ses conversations
- **Liées au dépôt** : fichier `.github/copilot-instructions.md` à la racine
- **Organisationnelles** : paramétrées pour harmoniser les réponses dans tous les projets

Génération de tests : Copilot peut créer des tests unitaires via des fichiers d'invite (`generate-unit-tests.prompt.md`) définissant un plan détaillé centré sur les fonctionnalités, la validation d'entrées et la gestion des erreurs.

Installation de GitHub Copilot

01

Visual Studio Code

Recherchez « GitHub Copilot » dans la marketplace VS Code, installez l'extension et connectez-vous avec votre compte GitHub. Une version gratuite est proposée pour les étudiants et mainteneurs open-source.

02

Visual Studio 2022

Copilot est intégré nativement depuis Visual Studio 2022. Activez-le via Extensions > Manage Extensions et connectez-vous avec vos identifiants GitHub.

03

Outils en Ligne

Sur GitHub.com, Copilot Chat est accessible via l'icône « Copilot » lors de la lecture d'un fichier de code dans l'interface web.

Astuce pour étudiants : Profitez de l'accès gratuit à Copilot en vérifiant votre statut étudiant via GitHub Education. Cela vous donne accès à toutes les fonctionnalités premium sans frais.

Exercices Pratiques avec Copilot

1

Découverte des Suggestions

Dans VS Code, créez un fichier `app.js`. Tapez un commentaire décrivant une fonction : `// fonction qui calcule la moyenne d'un tableau`. Laissez Copilot proposer le code correspondant. Acceptez ou parcourez les différentes suggestions avec Tab ou Ctrl +].

2

Génération de Tests Unitaires

Créez le fichier d'invite `generate-unit-tests.prompt.md` dans `.github/prompts`. Sélectionnez une fonction dans votre code et exécutez `/generate-unit-tests` pour générer des tests respectant le plan officiel.

3

Personnalisation via Instructions

Ajoutez des consignes dans `.github/copilot-instructions.md` telles que « Utiliser des retours précoces lorsque possible » ou « Répondre en français ». Copilot appliquera ces recommandations à toutes les suggestions du dépôt.

Génération de Tests Unitaires avec GitHub Copilot

⭐ Mode « Prompt Files » – recommandé par GitHub en 2025

GitHub Copilot permet aujourd'hui de **générer automatiquement des tests unitaires de grande qualité**, en suivant un **plan officiel robuste**, grâce à un fichier d'invite dédié (prompt file).

L'exécution se fait ensuite via la commande : `/generate-unit-tests`

Pourquoi utiliser cette méthode ?

Qualité Professionnelle

Suivi du plan officiel GitHub (core tests + validations + erreurs + side-effects)

Tests Reproductibles

Utile en certification (MSPR, projet DIADS) et audits de code

Bonnes Pratiques

Mocking, AAA pattern, test names clairs, couverture automatique

Productivité Maximale

Entre 10 et 20× plus rapide que l'écriture manuelle

Étapes de Configuration et Utilisation

01

Créer le fichier d'invite

Créez `.github/prompts/generate-unit-tests.prompt.md` dans votre projet. Ce fichier définit la stratégie de génération des tests.

02

Sélectionner la fonction

Ouvrez et sélectionnez la fonction à tester dans votre code. Copilot analysera les paramètres, la structure et les comportements attendus.

03

Exécuter la commande

Ouvrez Copilot Chat et tapez `/generate-unit-tests`. Vous pouvez préciser : `/generate-unit-tests function_name=calculate_total framework=pytest`

04

Réviser les tests générés

Copilot génère entre 5 et 8 tests complets couvrant : fonctionnalités principales, validation d'entrées, gestion d'erreurs et effets de bord.

Structure du fichier d'invite

Le fichier `generate-unit-tests.prompt.md` contient :

Stratégie de Tests

1. **Core Functionality Tests** - Comportement principal et valeurs de retour
2. **Input Validation Tests** - Types invalides, null/undefined, valeurs limites
3. **Error Handling Tests** - Exceptions attendues et messages d'erreur
4. **Side Effects Tests** - Appels externes, changements d'état, dépendances

Exigences de Structure

- Framework du projet (pytest, jest...)
- **AAA Pattern** : Arrange / Act / Assert
- Noms de tests descriptifs
- Regroupement avec describe/context
- Mocking propre des dépendances externes

Structure du Fichier generate-unit-tests.prompt.md

```
---
mode: 'agent'
description: 'Generate unit tests for selected functions or methods'
---

## Task
Analyze the selected function/method and generate focused unit tests that thoroughly validate its behavior.

## Test Generation Strategy

1. **Core Functionality Tests**
   - Test the main purpose/expected behavior
   - Verify return values with typical inputs
   - Test with realistic data scenarios

2. **Input Validation Tests**
   - Invalid input types
   - null / undefined values
   - empty strings/arrays/objects
   - boundary values (min/max...)

3. **Error Handling Tests**
   - Expected exceptions
   - Error messages
   - Edge cases

4. **Side Effects Tests**
   - External calls
   - State changes
   - Interaction with dependencies

## Test Structure Requirements
- Use the framework of the project (pytest, jest...)
- AAA Pattern : Arrange / Act / Assert
- Descriptive test names
- Group tests with describe/context blocks
- Mock external dependencies cleanly

Target function: ${input:function_name}
Testing framework: ${input:framework}
```

Ce fichier définit le plan officiel recommandé par GitHub pour garantir une couverture de tests complète et professionnelle. Les variables `${input:function_name}` et `${input:framework}` sont automatiquement remplies lors de l'exécution de la commande.

Exemple Complet : Du Code aux Tests

Code Source

```
def apply_discount(price, percent):
    if percent < 0 or percent > 100:
        raise ValueError("Invalid discount")
    return price * (1 - percent / 100)
```

Commande Copilot

```
/generate-unit-tests function_name=apply_discount
framework=pytest
```

Tests Générés Automatiquement

```
def test_apply_discount_valid_percent():
    assert apply_discount(100, 10) == 90

def test_apply_discount_zero_percent():
    assert apply_discount(100, 0) == 100

def test_apply_discount_full_percent():
    assert apply_discount(100, 100) == 0

def test_apply_discount_negative_percent():
    with pytest.raises(ValueError):
        apply_discount(100, -5)

def test_apply_discount_percent_above_100():
    with pytest.raises(ValueError):
        apply_discount(100, 150)
```

Résumé : Workflow Complet

1

1. Créer le fichier

.github/prompts/generate-unit-tests.prompt.md

2

2. Sélectionner

Ouvrir ou sélectionner une fonction

3

3. Lancer

/generate-unit-tests

4

4. Obtenir

Tests fonctionnels, validation, erreurs, effets de bord, mocking propre

Copilot respecte strictement : ✔️ AAA Pattern ✔️ Tests d'erreurs ✔️ Tests frontières ✔️ Valeurs réalistes ✔️ Noms explicites ✔️ Regroupement logique

Bonnes Pratiques avec Copilot

Contexte Pertinent

Ouvrez les fichiers utiles et fermez ceux qui ne le sont pas. Copilot analyse les fichiers ouverts pour proposer des suggestions plus justes et contextuellement appropriées.

Relecture Systématique

Copilot peut générer du code incorrect ou des failles de sécurité. Relisez, testez et refactorez le code avant de l'intégrer dans votre projet de production.

Instructions Claires

Rédigez vos commentaires ou requêtes de manière précise pour obtenir des résultats pertinents. Si la suggestion est inadaptée, reformulez votre demande avec plus de détails.

Ces pratiques garantissent une utilisation optimale de Copilot tout en maintenant la qualité et la sécurité de votre code. L'IA est un assistant, pas un substitut à votre expertise de développeur.

TabNine : L'Assistant Privé et Personnalisable

TabNine se distingue par son approche axée sur la confidentialité et la personnalisation avancée. Cet assistant de code accélère la programmation tout en préservant la confidentialité de votre code source.

Confidentialité Garantie

TabNine fonctionne localement et n'envoie pas le code source vers des serveurs externes par défaut. Les suggestions s'appuient sur des modèles entraînés sur du code sous licences permissives.

Personnalisation Avancée

Réglage de la longueur des réponses (mode concis ou complet), définition de comportements de chat adaptés (focus performances ou explications détaillées), et création de commandes personnalisées partagées.

Intégrations Multiples

Extension disponible pour VS Code, JetBrains, Visual Studio, Eclipse, Neovim et de nombreux IDEs. Version auto-hébergée proposée pour les entreprises.

Personnalisation TabNine : Commandes Partagées

TabNine introduit un système de **commandes personnalisées** permettant d'automatiser des workflows spécifiques à votre équipe. Ces commandes sont définies dans un fichier `.tabnine_commands` à la racine du projet.

Exemples de Commandes Utiles

- **Audit de sécurité** : analyse automatique des dépendances et recherche de fonctions dangereuses
- **Génération de tests** : création de tests suivant un canevas standardisé
- **Standardisation de la journalisation** : application cohérente des pratiques de logging
- **Refactoring selon conventions** : transformation du code selon les normes de l'équipe

Ces commandes peuvent être partagées avec toute l'équipe, garantissant une cohérence dans les pratiques de développement et augmentant l'adoption de l'outil en entreprise.

Comportements de Chat

Définissez des profils adaptés à différents types d'utilisateurs :

- **Débutant** : réponses détaillées avec explication des concepts
- **Expert** : suggestions concises et techniques
- **Architecte** : accent sur l'évolutivité et les patterns

> La personnalisation fine de TabNine permet à chaque équipe d'adapter l'IA à ses pratiques spécifiques, maximisant ainsi la productivité et l'adoption.

Installation et Configuration de TabNine

1

Installation de l'Extension

Dans VS Code, recherchez « TabNine » dans la marketplace et installez l'extension. L'installation est rapide et ne nécessite aucune configuration complexe.

2

Configuration Initiale

Exécutez la commande `TabNine: config` pour ouvrir le panneau de configuration. Connectez-vous ou créez un compte si nécessaire pour accéder aux fonctionnalités avancées.

3

Activation du Chat

Activez TabNine Chat dans les paramètres. Pour les entreprises, configurez l'option de déploiement privé via les instructions officielles pour garantir la confidentialité totale.

Exercices Pratiques avec TabNine

1

Tester les Complétions

Créez un fichier Python et commencez à écrire une fonction de tri. Observez les complétions contextuelles proposées par TabNine. Testez la différence entre le mode standard et le mode « Comprehensive » pour des réponses détaillées.

2

Personnaliser le Chat

Dans le panneau de configuration, définissez un comportement de chat adapté à un développeur débutant (réponses détaillées, explication des concepts) ou à un architecte logiciel (accent sur l'évolutivité).

3

Créer une Commande Personnalisée

Ajoutez dans `.tabnine_commands` une commande `check-security` qui exécute un audit simple : analyse des dépendances et recherche de fonctions dangereuses. Testez la commande dans TabNine Chat.

IntelliCode : L'Intelligence de Microsoft

IntelliCode est intégré par défaut à Visual Studio et Visual Studio Code. Il fournit des suggestions contextuelles basées sur l'analyse de milliers de projets open source, plaçant les méthodes les plus probables en premier.

Fonctionnalités Clés

- **Saisie semi-automatique contextuelle** : prédiction des méthodes ou propriétés les plus appropriées au contexte du code
- **Complétions de paramètres** : mise en avant des arguments les plus plausibles lors de l'appel d'une méthode
- **Complétions de lignes entières** : pour C# dans Visual Studio 2022+, IntelliCode propose des lignes complètes basées sur le contexte
- **Suggestions de répétition** : détection des actions répétitives et proposition de les appliquer ailleurs dans le code

Langages Supportés

IntelliCode prend en charge une large gamme de langages et technologies :

- C# et VB.NET
- C++ et C
- JavaScript et TypeScript
- Python
- XAML

L'extension est activée par défaut dans la plupart des workloads Visual Studio, offrant une expérience transparente sans configuration supplémentaire.

Utilisation d'IntelliCode

Installation

Installez Visual Studio 2022 (version Community gratuite) et sélectionnez un workload (ex. « Développement .NET »). IntelliCode est installé par défaut avec la plupart des workloads.

Utilisation

Ouvrez un projet C# ou C++. Lorsque vous tapez du code, observez les suggestions surlignées d'une étoile ⭐. Ces suggestions sont les plus probables selon le contexte. Appuyez sur Tab pour accepter.

Configuration

Explorez les options dans Outils > Options > IntelliCode pour activer ou désactiver les fonctionnalités en préversion et personnaliser l'expérience selon vos préférences.

Kite : Un Pionnier Aujourd'hui Abandonné

Historique

Kite était un pionnier de l'autocomplétion intelligente, principalement pour Python. L'outil fonctionnait localement et fournissait des suggestions de code, des signatures de fonctions et des snippets de documentation directement dans l'IDE.

Arrêt du développement : L'entreprise a cessé le développement en 2021 avec le message : « nous avons arrêté le logiciel et ne le supportons plus ».

Points Clés à Retenir

- **Suggestions contextuelles** : pour Python, Kite proposait des complétions précises et des extraits de documentation
- **Intégration IDE** : l'outil s'intégrait à VS Code, Atom, PyCharm et d'autres environnements
- **Traitement local** : les modèles fonctionnaient hors ligne, protégeant la confidentialité du code
- **Limitations** : prise en charge limitée des autres langages et aucun support actuel

Recommandation : Les développeurs cherchant un outil gratuit et local devraient se tourner vers des solutions maintenues comme TabNine ou Copilot plutôt que d'utiliser Kite.

Préparer Votre Environnement de Développement

1

Visual Studio Code

Téléchargez et installez la version stable de VS Code. Installez les extensions nécessaires (GitHub Copilot, TabNine). Configurez Git et un interpréteur Python/Node.js en fonction de vos projets.

2

Visual Studio 2022

Installez la version Community (gratuite). Sélectionnez les workloads « Développement .NET » et « Développement Web ». Assurez-vous que IntelliCode est activé dans les paramètres.

3

Autres IDEs

Pour JetBrains (PyCharm, IntelliJ), installez les plugins correspondants depuis le marketplace. Pour Neovim, suivez les instructions spécifiques de chaque outil (ex. `TabNine::install`).

Configuration des Outils d'IA

Une configuration appropriée des assistants de code est essentielle pour maximiser leur efficacité et garantir la cohérence dans votre équipe de développement.

1 Compte GitHub

Créez ou connectez un compte GitHub. Souscrivez à Copilot en profitant des remises pour étudiants et développeurs open source. Vérifiez votre éligibilité via GitHub Education.

2 Comptes TabNine

Inscrivez-vous sur TabNine pour bénéficier du mode Pro (gratuit la première semaine). Activez TabNine Chat pour accéder aux fonctionnalités conversationnelles avancées.

3 Instructions Personnalisées

Créez le fichier `.github/copilot-instructions.md` dans vos dépôts pour préciser les conventions (langage, style de code, frameworks). Pour TabNine, créez `.tabnine_commands` à la racine du projet.

4 Mises à Jour

Activez les mises à jour automatiques des extensions pour bénéficier des dernières améliorations et correctifs de sécurité. Vérifiez régulièrement les nouvelles fonctionnalités.

Méthodologie d'Utilisation des Suggestions

1 Définir le Besoin

Rédigez un commentaire clair décrivant l'objectif de la fonction ou du module à écrire. Les assistants utilisent ces commentaires pour comprendre l'intention et proposer du code approprié.

2 Observer les Suggestions

Acceptez, ignorez ou parcourez les différentes suggestions proposées par l'outil. Utilisez les raccourcis clavier (Tab, Ctrl +]) pour naviguer efficacement entre les options.

3 Adapter et Vérifier

Modifiez la suggestion pour l'adapter à votre cas d'utilisation spécifique, puis testez-la rigoureusement. Ajoutez des assertions, gérez les exceptions et respectez les bonnes pratiques de sécurité.

4 Documenter

Utilisez l'IA pour générer des commentaires ou de la documentation. TabNine et Copilot peuvent résumer des fonctions ou créer des docstrings conformes aux standards.

Analyse et Refactoring avec l'IA

Analyse de Code

TabNine Chat et Copilot Chat peuvent expliquer le comportement d'une fonction, identifier des bugs potentiels ou suggérer des améliorations de performance. Configurez TabNine pour fournir des explications détaillées ou axées sur l'architecture selon vos besoins.

Refactoring Intelligent

Demandez à l'IA de proposer une version plus lisible ou plus performante d'une fonction. Les suggestions doivent être confrontées aux conventions de votre équipe et validées par des tests avant intégration.

Audit de Sécurité

Utilisez des commandes personnalisées (TabNine) ou des instructions Copilot pour auditer votre code et repérer des failles. Par exemple, une commande `check-security` peut automatiser la recherche de vulnérabilités courantes :

- Analyse des dépendances obsolètes
- Détection de fonctions dangereuses
- Vérification des pratiques de sécurité
- Identification des données sensibles exposées

Cette approche proactive améliore significativement la qualité et la sécurité du code produit.

Génération de Tests et Documentation

Tests Unitaires avec Copilot

Suivez le guide « Generate unit tests » en créant un fichier d'invite `generate-unit-tests.prompt.md` dans `.github/prompts`. Ce fichier définit un plan détaillé pour générer des tests ciblés couvrant :

- La fonctionnalité principale
- La validation des entrées
- La gestion des erreurs
- Les effets de bord

Sélectionnez une fonction et exécutez `/generate-unit-tests` dans Copilot Chat pour obtenir des tests complets et bien structurés.

Documentation avec TabNine

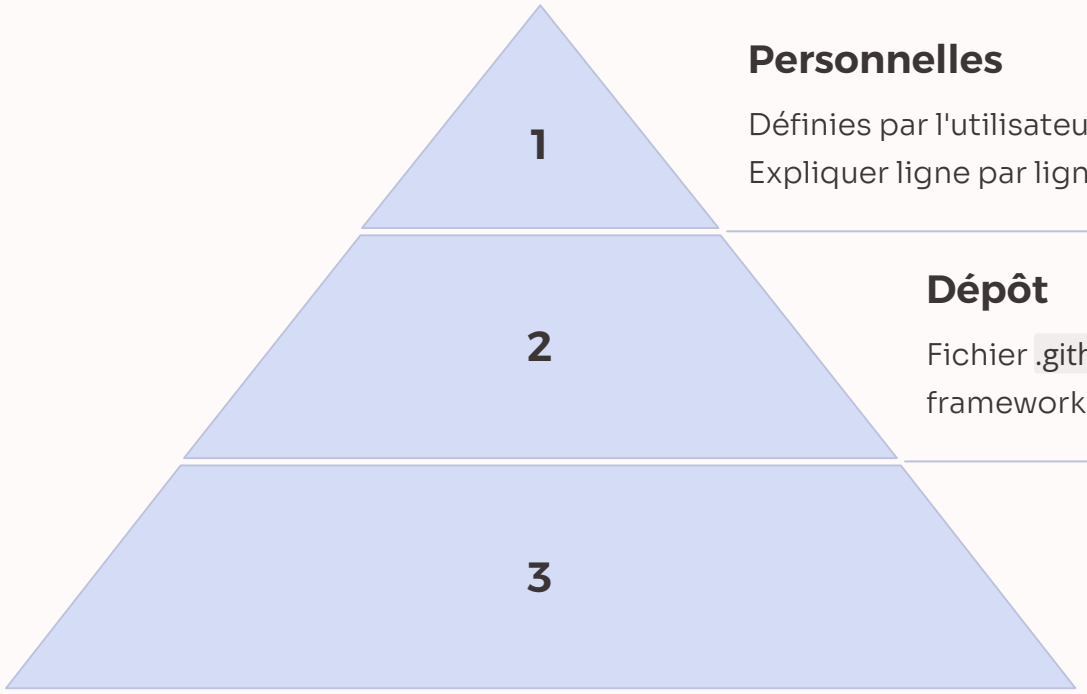
TabNine propose un « Documentation Agent » qui génère des commentaires et des descriptions de fonctions selon les standards de l'entreprise. Créez une commande personnalisée pour documenter automatiquement :

- Classes et interfaces
- Méthodes publiques
- APIs REST
- Modules complexes

Cette automatisation garantit une documentation cohérente et à jour, facilitant la maintenance et la collaboration en équipe.

Instructions Personnalisées pour Copilot

Copilot offre trois niveaux de personnalisation avec un système de priorité clair : instructions personnelles > instructions de chemin > instructions du dépôt > instructions d'organisation.



Personnelles

Définies par l'utilisateur pour toutes ses conversations (ex. « Répondre en français », « Expliquer ligne par ligne »)

Dépôt

Fichier `.github/copilot-instructions.md` appliqué à l'ensemble du dépôt (langage, frameworks, style de tests)

Organisation

Paramétrées par le propriétaire pour harmoniser les réponses dans tous les projets (langue officielle, règles de sécurité)

Les **fichiers d'invite** (`.github/prompts/*.prompt.md`) contiennent des prompts réutilisables pour la génération de tests, création de README, ou autres tâches récurrentes.

Projet 1 : Application Web E-Commerce

Objectif

Développer une application web simple avec catalogue de produits, panier d'achat et système de paiement en utilisant GitHub Copilot comme assistant principal.

Architecture Technique

- **Backend** : Node.js avec Express
- **Base de données** : SQLite ou MongoDB
- **Frontend** : React avec hooks
- **Tests** : Jest et React Testing Library

Utilisation de Copilot

Générez les routes Express, les modèles de base de données et les hooks React. Utilisez Copilot Chat pour expliquer les concepts et générer des tests unitaires via le prompt `generate-unit-tests`.

Personnalisation

Ajoutez une instruction de dépôt dans `.github/copilot-instructions.md` précisant l'utilisation de React et des hooks pour que Copilot propose des solutions alignées avec ces technologies.

Livrables

- Code fonctionnel et testé
- Tests unitaires complets
- README généré avec un prompt personnalisé
- Démonstration de l'application

Projet 2 : Tableau de Bord Analytique

Créez un tableau de bord pour visualiser et analyser des données utilisateur en exploitant les capacités de TabNine pour le développement Python et JavaScript.

1

Technologies

Backend Python avec FastAPI, analyse de données avec Pandas et Matplotlib. Frontend JavaScript avec D3.js ou Chart.js pour les visualisations interactives.

2

Utilisation de TabNine

Profitez des complétions intelligentes en Python (manipulation de DataFrames, génération de graphiques) et JavaScript. Testez les modes concis/complet pour obtenir des explications de code adaptées à votre niveau.

3

Personnalisation

Créez un comportement de chat qui fournit des explications didactiques sur les statistiques pour les étudiants non spécialistes. Définissez des commandes pour générer des graphiques types (histogrammes, courbes temporelles).

4

Livrables

Application déployée, code documenté, tests automatisés et rapport détaillé sur l'utilisation des outils d'IA et leur impact sur la productivité.

Projet 3 : API REST de Gestion de Contenu

Objectif et Technologies

Développer une API REST pour un système de gestion de contenu (CMS) en utilisant IntelliCode comme assistant principal. Choisissez entre Node.js/Express ou .NET Core selon vos préférences.

Implémentez les opérations CRUD complètes (create, read, update, delete) sur des articles avec validation, authentification et gestion des erreurs.

Utilisation d'IntelliCode

Laissez IntelliCode proposer les méthodes et paramètres les plus probables grâce aux complétions étoilées ⭐. Pour le C#, observez les complétions de lignes entières et intégrez les suggestions pertinentes.

Personnalisation

Dans Visual Studio, activez les fonctionnalités en préversion pour tester les complétions automatiques avancées. Utilisez Copilot ou TabNine en complément pour générer des tests ou des scripts frontaux.

Livrables Attendus

- API fonctionnelle avec tous les endpoints CRUD
- Documentation Swagger/OpenAPI complète
- Tests automatisés (unitaires et d'intégration)
- Démonstration de l'utilisation des suggestions IntelliCode
- Rapport sur l'impact de l'IA sur le développement

Évaluation et Ressources

Critères d'Évaluation

L'évaluation se fait via un projet professionnel intégrant back-end métier, nettoyage et visualisation de données :

- **Pertinence de l'utilisation des assistants** : exploitation des suggestions, personnalisation des outils, génération de tests
- **Qualité du code** : clarté, respect des conventions, absence de vulnérabilités
- **Couverture de tests** : tests unitaires et d'intégration générés et exécutés
- **Documentation** : README clair, commentaires, instructions de déploiement
- **Présentation orale** : explication des choix techniques, apports de l'IA, limites et mesures qualité

Ressources Complémentaires

Documentation officielle :

- GitHub Docs – Qu'est-ce que GitHub Copilot ?
- GitHub Docs – Responsabilité et sécurité
- Docs – Personnalisation de la réponse
- TabNine Docs – Overview
- Microsoft Learn – IntelliCode

Plateformes d'apprentissage :

- 360 Learning : modules sur les assistants de code
- LinkedIn Learning : « AI Coding Assistant Essentials »
- Bibliothèque ENI : ouvrages sur l'IA et la programmation