Leitfaden zur KI-gestützten Softwareentwicklung: Effizienz und Innovation im Semesterprojekt

Versionshistory

Version	Datum	Autor	Beschreibung
1.0	05.09.2025	GitHub Copilot	Ersterstellung des Dokuments

Inhaltsverzeichnis

1. Vorwort: Ihr KI-Co-Pilot für das Semesterprojekt

Dieses Dokument ist ein Begleiter zum Hauptskript `SYP5_Lernskript-Semesterprojekt-Scrumban-CleanArchitecture.md`. Es zeigt, wie die im SEM-Unterricht gelernten KI-Konzepte praktisch im Semesterprojekt angewendet werden können. Das Ziel ist, die Produktivität des Teams durch den schrittweisen und gezielten Einsatz von KI-Werkzeugen zu steigern – von der einfachen Code-Vervollständigung bis hin zur automatisierten Erstellung von Features.

2. Projektstart & Grundlagen der KI-Assistenz

Fokus: Einrichtung und erste Produktivitätsgewinne

In diesem Kapitel legen wir den Grundstein für die KI-gestützte Softwareentwicklung. Wir richten unsere Entwicklungsumgebung ein, lernen die Kernfunktionen von GitHub Copilot kennen und entdecken, wie wir durch effektive Kommunikation und Personalisierung das Maximum aus unserem KI-Assistenten herausholen.

2.1. Die intelligente Werkbank: AI Assisted IDEs

Moderne integrierte Entwicklungsumgebungen (IDEs) sind nicht mehr nur Texteditoren mit Syntax-Hervorhebung. Sie entwickeln sich zu intelligenten Werkbänken, die uns aktiv bei der Code-Erstellung unterstützen. Diese "AI Assisted IDEs" integrieren KI-Funktionen nahtlos in den Entwickler-Workflow.

2.1.1. VSCode mit GitHub Copilot als Basis

Visual Studio Code (VSCode) hat sich als eine der populärsten IDEs etabliert. In Kombination mit **GitHub Copilot**, einem KI-gestützten "Pair Programmer", wird es zu einem mächtigen Werkzeug für die moderne Softwareentwicklung.

Installation und Konfiguration: Die Einrichtung ist unkompliziert. Nach der Installation von VSCode kann die GitHub Copilot-Erweiterung direkt aus dem Marketplace hinzugefügt werden. Eine Anmeldung mit einem GitHub-Konto, das für Copilot freigeschaltet ist, genügt, um zu starten.

Kernfunktionen:

1. **Intelligente Code-Vervollständigung (Code Completion):** Während Sie tippen, schlägt Copilot nicht nur einzelne Wörter oder Zeilen, sondern ganze Codeblöcke oder Funktionen vor. Diese Vorschläge basieren auf dem Kontext Ihres Projekts und dem Code, den Sie gerade schreiben. Sie können Vorschläge mit der `Tab`-Taste annehmen oder mit `Alt+]` und `Alt+[` durch Alternativen blättern.

   Beispiel (Dart/Flutter): Sie beginnen, ein einfaches `StatelessWidget` zu definieren, und Copilot vervollständigt die gesamte Klasse. Sie tippen: `class MyButton extends StatelessWidget` und Copilot schlägt vor:

   ```
   class MyButton extends StatelessWidget {
     final String text;
     final VoidCallback onPressed;

     const MyButton({
       Key? key,
       required this.text,
       required this.onPressed,
     }) : super(key: key);

     @Override
     Widget build(BuildContext context) {
       return ElevatedButton(
         onPressed: onPressed,
   ```

```
        child: Text(text),
      );
    }
  }
```

2. **Chat-basierte Interaktion (Chat-Modi):** Über die Chat-Ansicht (`Ctrl+Alt+I`) können Sie in natürlicher Sprache mit Copilot kommunizieren. Der Chat bietet verschiedene Modi, die für spezifische Aufgaben optimiert sind:
    - `/ask` **(Fragen):** Dies ist der Standardmodus für allgemeine Fragen. Nutzen Sie ihn, um Code zu analysieren, Erklärungen zu erhalten oder allgemeine Programmierfragen zu stellen, ohne den Code direkt zu verändern.
    - `/edit` **(Bearbeiten):** Dieser Modus ist für direkte Code-Änderungen gedacht. Sie können einen Code-Block markieren und Copilot anweisen, ihn zu bearbeiten (z.B. "Refactor this into a separate function" oder "Add error handling"). Die Änderungen werden als Diff angezeigt, bevor Sie sie übernehmen.
    - `@workspace` **(Agent):** Für größere, dateiübergreifende Aufgaben agiert Copilot als autonomer Agent. Geben Sie ein übergeordnetes Ziel vor (z.B. "@workspace Erstelle eine Task-Manager-App mit der Möglichkeit, Aufgaben hinzuzufügen, zu löschen und als erledigt zu markieren"), und der Agent plant und implementiert die Lösung selbstständig über mehrere Dateien hinweg.
3. **Inline-Chat:** Direkt im Editor können Sie einen Code-Abschnitt markieren und mit `Ctrl+I` einen Inline-Chat starten. Dies ist ideal für gezielte Anpassungen, Refactoring oder das Hinzufügen von Fehlerbehandlung zu einem spezifischen Code-Block.

Quellen:

- Get started with GitHub Copilot in VS Code
- GitHub Copilot in VS Code
- What are the best AI code assistants for vscode in 2025?
- Use chat modes in VS Code

2.1.2. Effektive Kommunikation mit der KI: Grundlagen des Context Engineering

Um präzise und nützliche Ergebnisse von einer KI zu erhalten, reicht es nicht aus, nur eine Frage zu stellen. Die Qualität der Antwort hängt entscheidend von der Qualität des Kontexts ab, den wir der KI zur Verfügung stellen. Dieser Prozess wird als **Context Engineering** bezeichnet.

Was ist Context Engineering und warum ist es entscheidend? Context Engineering ist die Kunst und Wissenschaft, das "Kontextfenster" eines Sprachmodells (LLM) mit genau den richtigen Informationen für eine bestimmte Aufgabe zu füllen. Es geht weit über das reine "Prompt Engineering" (die Formulierung der Frage) hinaus und umfasst alle Informationen, die das Modell sieht, bevor es eine Antwort generiert.

Der Kontext kann umfassen:

- **System-Anweisungen:** Grundlegende Verhaltensregeln für die KI.
- **Benutzer-Prompt:** Die eigentliche Frage oder Aufgabe.
- **Kurzzeitgedächtnis:** Der bisherige Gesprächsverlauf.

- **Langzeitgedächtnis:** Wissen aus früheren Interaktionen.
- **Abgerufene Informationen (RAG):** Externe Daten aus Dokumenten, Datenbanken oder APIs.
- **Verfügbare Werkzeuge (Tools):** Definitionen von Funktionen, die die KI aufrufen kann.

Ein Fehler der KI ist oft kein Fehler des Modells selbst, sondern ein "Kontextfehler" – die KI hatte nicht die nötigen Informationen, um die Aufgabe korrekt zu lösen.

Praxis-Tipps für gute Prompts:

1. **Setzen Sie ein übergeordnetes Ziel:** Beginnen Sie mit einer allgemeinen Beschreibung dessen, was Sie erreichen möchten, besonders bei einer leeren Datei. Das "stimmt" die KI auf die Aufgabe ein. *Beispiel für Boilerplate-Code (Clean Architecture):*

   ```
   // Erstelle die grundlegende Ordnerstruktur und die Basisklassen für eine
   Clean Architecture in Flutter.
   // Lege die Layer 'domain', 'data' und 'presentation' an.
   // Erstelle im 'domain'-Layer ein Beispiel-Entity 'Product' mit 'id', 'name'
   und 'price'.
   // Erstelle außerdem ein abstraktes 'ProductRepository'-Interface im
   'domain'-Layer.
   ```

2. **Formulieren Sie einfach und spezifisch:** Brechen Sie komplexe Aufgaben in kleine, logische Schritte herunter. Lassen Sie die KI nach jedem Schritt Code generieren, anstatt alles auf einmal zu verlangen.

   Beispiel (Java/Spring Boot): Statt "Erstelle eine komplexe Benutzer-API" gehen Sie schrittweise vor:

   1. Prompt 1: `// Erstelle eine Spring Boot REST-Controller-Klasse 'UserControler' mit einer GET-Methode für '/api/users', die eine statische Liste von User-DTOs zurückgibt.`
   2. (Nachdem der Code generiert wurde) Prompt 2: `// Erweitere den Controller um eine GET-Methode für '/api/users/{id}', die einen einzelnen Benutzer anhand seiner ID aus der Liste sucht.`
   3. Prompt 3: `// Füge eine POST-Methode für '/api/users' hinzu, um einen neuen Benutzer zu erstellen. Verwende @RequestBody und validiere mit @Valid, dass Name und E-Mail des Benutzers nicht null sind.`

3. **Geben Sie Beispiele:** "One-shot" oder "Few-shot" Learning, bei dem Sie der KI ein oder mehrere Beispiele für das gewünschte Ergebnis geben, ist extrem wirkungsvoll. *Beispiel (Flutter/Dart):*

   ```
   // Konvertiere die folgende Liste von Maps in eine Liste von 'Product'-
   Widgets.
   // Beispiel: Aus [{'name': 'Apfel', 'category': 'Frucht'}] soll ein
   `ProductTile(name: 'Apfel', category: 'Frucht')` werden.
   final productsData = [{'name': 'Karotte', 'category': 'Gemüse'}, {'name':
   'Banane', 'category': 'Frucht'}];
   // Erwartetes Ergebnis ist eine `List<ProductTile>`.
   ```

4. **Halten Sie relevante Dateien offen:** Copilot nutzt die "Neighboring Tabs"-Technik, um den Kontext aus anderen geöffneten Dateien zu beziehen.
5. **Verwenden Sie gute Coding-Praktiken:** Aussagekräftige Variablen- und Funktionsnamen helfen nicht nur Menschen, sondern auch der KI, Ihren Code zu verstehen.

Quellen:

- Context Engineering - What it is, and techniques to consider
- The New Skill in AI is Not Prompting, It's Context Engineering
- How to write better prompts for GitHub Copilot
- AI Agent Best Practices: 12 Lessons from AI Pair Programming for Developers

---

2.1.3. Den Copiloten personalisieren: Customizing in VSCode

Um Copilot optimal an die Anforderungen Ihres Projekts und Ihren persönlichen Stil anzupassen, bietet VSCode mächtige Personalisierungsoptionen.

Custom Instructions

`Custom Instructions` müssen in folgender Datei: `.github/copilot-instructions.md)` definiert werden.

Mit *Custom Instructions* können Sie Copilot eine feste Rolle und dauerhaften Kontext für ein Projekt geben. Anstatt bei jeder Anfrage dieselben Regeln zu wiederholen, definieren Sie diese einmalig in einer Markdown-Datei.

- **Anwendung:** Erstellen Sie eine Datei namens `copilot-instructions.md` im `.github`-Verzeichnis Ihres Projekts.
- **Inhalt:** Definieren Sie hier allgemeine Richtlinien, z.B. Programmierstil, Namenskonventionen oder bevorzugte Technologien. *Beispiel (Java/Spring):*

  ```
  # Projekt-Richtlinien für Spring Boot
  - Alle Klassen im Service-Layer müssen mit @Service annotiert sein.
  - Abhängigkeiten sollen über Konstruktor-Injection injiziert werden.
  - Verwende JPA und Spring Data für den Datenbankzugriff.
  - Definiere DTOs (Data Transfer Objects) für die API-Kommunikation, um
  Entitäten nicht direkt preiszugeben.
  - Nutze das "Builder"-Entwurfsmuster für die Erstellung von komplexen
  Objekten.
  ```

- **Selektive Anwendung:** Für spezifischere Regeln können Sie mehrere `.instructions.md`-Dateien erstellen und deren Gültigkeitsbereich über einen `applyTo`-Header steuern. Dies ist nützlich, um unterschiedliche Anweisungen für verschiedene Teile Ihres Projekts zu definieren, z.B. für Tests, Backend- oder Frontend-Code.

  Beispiel für JUnit-Test-spezifische Anweisungen:

Angenommen, Sie erstellen eine Datei `.github/instructions/junit-rules.instructions.md`:

```
---
description: "Regeln für JUnit 5 Tests"
applyTo: "src/test/java/**/*.java"
---
# Anweisungen für JUnit-Testdateien

Du bist ein Java-Test-Experte, der sich auf das JUnit 5-Framework
spezialisiert hat.

- Verwende die Assertions aus `org.junit.jupiter.api.Assertions`.
- Strukturiere Tests mit `@Nested`-Klassen und `@DisplayName` für bessere
Lesbarkeit.
- Mocke Abhängigkeiten mit dem Mockito-Framework, z.B. mit `@Mock` und
`MockitoAnnotations.openMocks(this)`.
- Jede Testmethode muss mit `@Test` annotiert sein.
- Generiere für jede Methode aussagekräftige Testfälle, die sowohl Erfolgs-
als auch Fehlerfälle abdecken.
```

Diese Regeln werden von Copilot nun automatisch nur dann angewendet, wenn Sie in einer Java-Datei unter `src/test/java/` arbeiten.

Custom Chatmodes

Custom Chat Modes sind vordefinierte Konfigurationen, die das Verhalten des Chats für spezielle Aufgaben optimieren. Sie können damit eigene "Experten-Modi" definieren.

- **Anwendung:** Erstellen Sie eine `.chatmode.md`-Datei (z.B. `CodeReviewer.chatmode.md`) im `.github/chatmodes`-Verzeichnis.
- **Inhalt:** Eine Chat-Mode-Datei besteht aus:
  - **Frontmatter (Header):** Eine Beschreibung, eine Liste der erlaubten `tools` (z.B. nur lesende Tools für einen Reviewer) und optional ein bestimmtes KI-`model`.
  - **Body (Anweisungen):** Detaillierte Anweisungen, die die Rolle und das Verhalten der KI in diesem Modus definieren.
- **Beispiel: Ein "Code Reviewer"-Chatmode**

  Die Datei `/.github/chatmodes/CodeReviewer.chatmode.md` könnte so aussehen:

```
---
description: 'Überprüft den Code auf Qualität und Best Practices, ohne ihn zu
ändern.'
tools: ['codebase', 'problems', 'usages']
---
# Code Reviewer Modus

Du bist ein erfahrener Senior-Entwickler, der einen gründlichen Code-Review
```

```
durchführt.
Deine Aufgabe ist es, den Code auf Qualität, Best Practices und die
Einhaltung der [Projekt-Richtlinien](../copilot-instructions.md) zu prüfen.

## Analysefokus
- Identifiziere potenzielle Bugs, Sicherheitslücken oder Performance-
Probleme.
- Bewerte die Lesbarkeit, Wartbarkeit und Struktur des Codes.
- Gib konstruktives, spezifisches Feedback mit klaren Erklärungen.

## Wichtige Regeln
- Schreibe oder ändere **niemals** selbst Code.
- Konzentriere dich darauf zu erklären, **was** geändert werden sollte und
**warum**.
- Schlage alternative Ansätze vor, wenn es sinnvoll ist.
```

Wenn Sie diesen Modus aktivieren, wird Copilot den Code analysieren und Feedback geben, anstatt direkte Code-Änderungen vorzuschlagen.

Durch die Kombination dieser Techniken verwandeln Sie GitHub Copilot von einem allgemeinen Assistenten in einen hochspezialisierten Co-Piloten, der genau auf Ihr Projekt und Ihre Arbeitsweise zugeschnitten ist.

Quellen:

- Use custom instructions in VS Code
- Use chat modes in VS Code

---

3. Fortgeschrittene Techniken, Tooling & Wissensmanagement

Fokus: Tools und Projektwissen für die KI nutzbar machen

3.1. Bereitstellung von Tools für KI-Agenten (LLM's) mittels MCP

3.1.1. Was ist MCP?

- Defakto Standard für die Verwendung von Tools durch LLM's
- Aufbau und Funktionsweise

3.1.2. Einsatz von MCP-Server in der KI - gestützten SW-Entwicklung

- MCP - Repositories (Übersicht)
- Nützliche MCP-Servers in der SW-Entwicklung

3.1.3. Konfiguration und Integration von MCP-Server in KI Assisted IDE's

- Konfiguration in den unterschiedlichen IDE's (VSCode, Windurf, ClaudCode, ...)
- Konfigurationsarten (Cloud, lockal, Docker)

3.1.4. Verwendung von MCP-Server in VSCode

- lokale Konfiguration von MCP-Server
- Aktivieren deaktiveren von Server und Tools
- Defintion der verwendbaren (MCP) Tools in custom chatmodes

3.2. Das Projektgedächtnis: Retrieval Augmented Generation (RAG)

3.2.1. Was ist RAG?

- Die Idee einfach erklärt: Wie eine KI lernt, projektspezifische Dokumente zu "lesen".

3.2.2. KI mit Projekt-Kontext: RAG im Einsatz

4. Automatisierung & Agenten-basierte Entwicklung

`Fokus: Von der Assistenz zur autonomen Ausführung zum Beispiel durch Plattformen wie n8n

4.1. Definition von automatisierten Workflows

4.1.1. Projektmanagement-Workflows

Wie man Copilot nutzt, um das Scrumban-Board (GitHub Projects/Trello) zu pflegen (z.B. User Stories erstellen, Tasks ableiten).

4.1.2. Entwicklungs-Workflows

Definition von Abläufen für wiederkehrende Aufgaben (z.B. "Erstelle Feature von API bis Datenbank").

4.2. Erstellung von spezialisierten KI-Agenten

4.2.1. Was ist ein Agent?

Der Unterschied zwischen einem Chatbot und einem autonomen Helfer.

4.2.2. Werkzeuge zur Agenten-Erstellung (z.B. n8n)

- Übersicht über aktuelle Werkzeuge/Frameworks zur Erstellung von Agenten (z.B. n8n)

4.2.3. Praxis-Beispiel: Ein "Bug-Analyse-Agent"

```
- Der Agent nimmt eine Fehlermeldung entgegen.
- Er durchsucht den MCP-Server nach ähnlichen, bereits gelösten Problemen.
- Er schlägt eine Lösung oder den zuständigen Entwickler vor.
```

4.2.4. Die BMAD-Methode: Struktur für die Agenten-Entwicklung

- Einführung in die "Build, Measure, Analyze, Decide"-Methode als strukturierter Ansatz, um eigene KI-Lösungen und Agenten zu entwickeln und zu verbessern.

4.3. Glossar

Hier werden alle wichtigen Begriffe (MCP-Server, RAG, Context Engineering, Agent, BMAD etc.) kurz und verständlich erklärt.