KI-gestützte Software-Entwicklung

Inhaltsverzeichnis

Dokumenthistorie

Datum	Änderung	Autor
15.12.2025	Erstversion	KUW
16.12.2025	Erweiterung um Kapitel 1.6.5 "Komposition von Agentic Workflows"	KUW
16.12.2025	Erweiterung um Kapitel 1.2.4 "Technische Funktionsweise (Prompt-Architektur)"	KUW

1. KI-gestützte Software-Entwicklung mit VSCode und AntiyGravity

1.1. Zielsetzung

Dieses Lernscript behandelt den professionellen Einsatz von KI-Coding-Assistenten in der modernen Software-Entwicklung. Der Fokus liegt auf zwei führenden Plattformen:

1. **VSCode + GitHub Copilot** – Der etablierte Standard für KI-gestützte Entwicklung
2. **Google Antigravity** – Der fortschrittliche agentenbasierte Ansatz

Das Script vermittelt nicht nur die grundlegenden Funktionen, sondern zeigt insbesondere das **volle Potenzial** dieser Werkzeuge durch:

- **Custom Instructions** für konsistente, projektspezifische KI-Antworten
- **Custom Agents** für spezialisierte Entwicklungsrollen
- **Workflows** für automatisierte, wiederkehrende Aufgaben
- **Parallele Delegation** für effiziente Team-Entwicklung

> 💡 **Lernziel:** Nach Durcharbeitung dieses Scripts können Sie KI-Assistenten optimal für Ihre Projektanforderungen konfigurieren und nutzen.

1.2. Einführung in KI-gestützte Software-Entwicklung

1.2.1. Überblick über KI-Coding-Assistenten

KI-Coding-Assistenten haben die Art und Weise, wie Software entwickelt wird, grundlegend verändert. Sie unterstützen Entwickler bei nahezu allen Aspekten des Entwicklungsprozesses – von der Codegenerierung über Refactoring bis hin zur Dokumentation.

Was können moderne KI-Assistenten?

Fähigkeit	Beschreibung
Code-Generierung	Erstellen von Code basierend auf natürlichsprachlichen Beschreibungen
Code-Vervollständigung	Intelligente Inline-Vorschläge während des Tippens
Refactoring	Umstrukturierung von Code unter Beibehaltung der Funktionalität
Debugging	Identifikation und Behebung von Fehlern
Dokumentation	Automatische Erstellung von Kommentaren und Dokumentation
Testing	Generierung von Unit- und Integrationstests
Code-Review	Analyse und Verbesserungsvorschläge für bestehenden Code

1.2.2. VSCode + GitHub Copilot vs. Antigravity

Die beiden führenden Plattformen verfolgen unterschiedliche Ansätze:

VSCode + GitHub Copilot

Philosophie: Erweiterung der IDE um KI-Fähigkeiten mit Chat-Interface und Inline-Suggestions.

Stärken:

- Tiefe Integration in VSCode-Ökosystem
- Umfangreiche Extension-Unterstützung
- **Custom Agents** mit sequentieller Verkettung via Handoffs
- Flexible Tool-Integration (MCP-Server, Extensions)

Kernkonzepte:

- `.agent.md` Dateien für spezialisierte Agents
- `.instructions.md` für projektspezifische Anweisungen
- Tool-Auswahl pro Agent konfigurierbar

Google Antigravity

Philosophie: Vollständig agentenbasierter Ansatz mit strukturierten Arbeitsmodi.

Stärken:

- Klare Trennung in **Planning**, **Execution** und **Verification** Modi
- Automatische Artifact-Erstellung (Tasks, Pläne, Walkthroughs)
- Parallele Tool-Ausführung für Effizienz
- Integrierte Browser-Steuerung und Screenshot-Fähigkeiten

Kernkonzepte:

- `task_boundary` für strukturierte Arbeitsabläufe
- `.agent/workflows/` für wiederverwendbare Prozesse
- `.gemini/` Ordner für projektspezifische Konfiguration

1.2.3. Grundlegende Konzepte

Context Engineering

Definition: Die Kunst, dem KI-Assistenten den optimalen Kontext bereitzustellen, um qualitativ hochwertige Antworten zu erhalten.

Elemente des Kontexts:

- **Workspace-Struktur:** Welche Dateien sind relevant?
- **Aktive Dokumente:** Was bearbeitet der Entwickler gerade?
- **Projekt-Metadaten:** Welche Technologien und Patterns werden verwendet?
- **Custom Instructions:** Welche Coding-Standards gelten?

> ⚠️ **Wichtig:** Die Qualität der KI-Antworten hängt direkt von der Qualität des bereitgestellten Kontexts ab.

Prompt Engineering

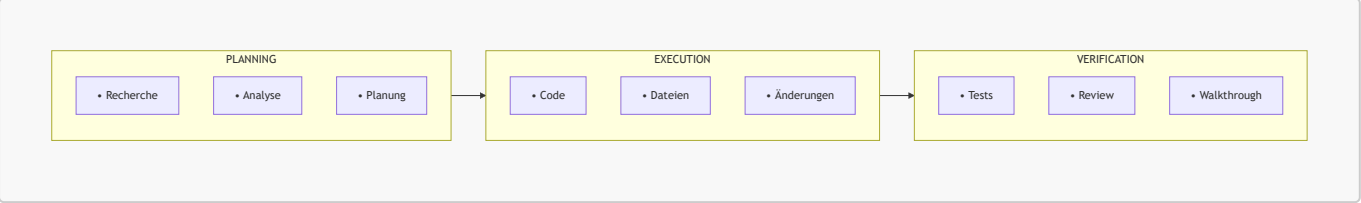
Definition: Die Formulierung von Anfragen, die zu präzisen und nützlichen Antworten führen.

Best Practices:

1. **Spezifisch sein:** Statt "Verbessere diesen Code" → "Refaktoriere diese Funktion für bessere Lesbarkeit unter Verwendung von Early Returns"
2. **Kontext mitliefern:** Relevante Constraints und Anforderungen nennen
3. **Schrittweise vorgehen:** Komplexe Aufgaben in Teilaufgaben zerlegen

Agent-Modi

Moderne KI-Assistenten arbeiten in verschiedenen Modi, die das Verhalten und die verfügbaren Tools bestimmen:



1.2.4. Technische Funktionsweise (Prompt-Architektur)

Um Copilot effektiv anzupassen, ist ein Verständnis der Kommunikation mit dem Large Language Model (LLM) unerlässlich. Jede Anfrage sendet ein komplexes **Kontextfenster** an das Modell, welches sich aus dem **System Prompt** und dem **User Prompt** zusammensetzt.

Der System Prompt

Der System Prompt definiert die grundlegenden Regeln und die Identität des Agenten. Er teilt dem LLM mit, *wie* es sich zu verhalten hat, und besteht aus vier Hauptabschnitten:

1. **Kernidentität & Globale Regeln:** Definiert die allgemeine Rolle des Agenten (z. B.: „Du bist ein intelligenter AI Coding Assistant“).
2. **Modellspezifische Anweisungen:** Regeln, die spezifische Eigenschaften oder Schwächen des verwendeten LLM korrigieren.
3. **Anweisungen zur Werkzeugnutzung:** Erklärt dem Agenten die Verwendung interner Tools (wie Editor, Terminal etc.).
4. **Anweisungen zum Ausgabeformat:** Legt fest, wie die Antwort für eine korrekte Verarbeitung formatiert sein muss.

Der User Prompt

Der User Prompt enthält die spezifischen Informationen der aktuellen Anfrage und Umgebung:

1. **Umgebungsinformationen:** Details zum Betriebssystem.
2. **Workspace-Informationen:** Textdarstellung der Projektstruktur (Ordner und Dateien).
3. **Kontextinformationen:** Aktueller Zeitstempel, Liste offener Terminals.
4. **Editor-Kontext:** Inhalt manuell hinzugefügter Dateien.
5. **Benutzeranfrage:** Die eigentliche Nachricht des Nutzers.

Diese Kette bildet das Kontextfenster für jede Interaktion.

Prompt-Injection der Komponenten

Die verschiedenen Anpassungsmöglichkeiten greifen an unterschiedlichen Stellen in diesen Prozess ein:

1. Custom Instructions:

- Dienen als **Projekt-Kontext**.
- Werden an das **Ende des System Prompts** angehängt.
- Haben hohe Priorität, da sie als letzte System-Anweisung stehen.

2. Prompt Files:

- Sind **wiederverwendbare Tasks**.
- Werden am **Anfang des User Prompts** eingefügt.
- Stehen noch **vor** dem Workspace-Kontext, um Aufgaben klar zu definieren (vermeidet Context Rot).

3. Custom Agents:

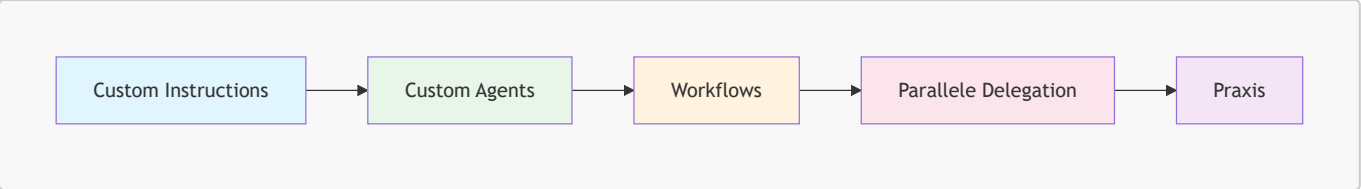
- Definieren **Identität und Workflow**.
- Stehen am **Ende des System Prompts**, sogar **nach** den Custom Instructions.
- Überschreiben damit ggf. Standardverhalten am stärksten.

1.2.5. Kapitelübersicht

Dieses Script ist in sechs aufeinander aufbauende Kapitel gegliedert:

Kapitel	Thema	Bedeutung
2	**Custom Instructions**	Grundlage für konsistente KI-Antworten – definiert projektweite Standards und Coding-Guidelines
3	**Custom Agents**	Spezialisierte KI-Rollen für unterschiedliche Aufgaben (Requirements, Architektur, Testing)
4	**Workflows**	Automatisierung wiederkehrender Prozesse – von Feature-Entwicklung bis Deployment
5	**Parallele Delegation**	Effiziente Nutzung mehrerer Agents für komplexe Aufgaben
6	**Praktische Beispiele**	Konkrete, sofort einsetzbare Konfigurationen und Vorlagen

Lernpfad



Empfehlung: Arbeiten Sie die Kapitel sequentiell durch, da jedes auf den vorherigen aufbaut. Die praktischen Beispiele in Kapitel 6 setzen das Verständnis aller vorherigen Konzepte voraus.

1.2.6. Quellen und Referenzen

Quelle	Beschreibung	Link
GitHub Copilot Docs	Offizielle Dokumentation	docs.github.com/copilot
VS Code AI Docs	VS Code KI-Features	code.visualstudio.com/docs/copilot
Prompt Engineering Guide	Best Practices für Prompts	github.com/dair-ai/Prompt-Engineering-Guide
Context Engineering	Simon Willison's Blog	simonwillison.net
Awesome Copilot	Community-Ressourcen	github.com/github/awesome-copilot

1.3. Custom Instructions

1.3.1. Was sind Custom Instructions?

Definition: Custom Instructions sind vordefinierte Anweisungen in Markdown-Dateien, die automatisch an jeden Chat-Request angehängt werden. Sie steuern, wie der KI-Assistent Code generiert und auf Anfragen reagiert.

Warum Custom Instructions verwenden?

Problem ohne Instructions	Lösung mit Instructions
Jedes Mal Coding-Standards erklären	Standards werden automatisch angewendet
Inkonsistente Antworten	Einheitlicher Stil im gesamten Projekt
Wiederholte Kontext-Eingabe	Einmalige Definition, dauerhafte Wirkung
Team-Mitglieder mit unterschiedlichen Prompts	Gemeinsame Standards für alle

> 💡 **Merke:** Custom Instructions gelten für **Chat-Interaktionen**, nicht für Inline-Suggestions während des Tippens.

1.3.2. Typen von Instruction-Dateien

VS Code unterstützt drei verschiedene Typen von Instruction-Dateien:

Übersicht

Datei	Anwendungsbereich
`.github/copilot-instructions.md`	Alle Requests im Workspace
`.instructions.md` (mit applyTo)	Spezifische Dateitypen/Pfade
`AGENTS.md`	Multi-Agent Workspaces

`.github/copilot-instructions.md`

Eigenschaften:

- Eine einzelne Datei im Workspace-Root
- Gilt automatisch für **alle** Chat-Requests
- Ideal für projektweite Coding-Standards

Einrichtung:

1. Aktivieren Sie die Einstellung `github.copilot.chat.codeGeneration.useInstructionFiles`
2. Erstellen Sie `.github/copilot-instructions.md`
3. Schreiben Sie Ihre Anweisungen in natürlicher Sprache

Beispiel:

```
## Projekt-Coding-Standards

### Allgemeine Regeln
- Bevorzuge objektorientierte Programmierung
- Schreibe aussagekräftige Variablennamen

### Error Handling
- Verwende try/catch für async Operationen
- Logge Fehler mit kontextbezogenen Informationen

### Testing
- Schreibe Unit-Tests für alle öffentlichen Funktionen
- Verwende das AAA-Pattern (Arrange, Act, Assert)
```

`.instructions.md` Dateien

Eigenschaften:

- Mehrere Dateien möglich
- Bedingte Anwendung via `applyTo` Glob-Pattern
- Speicherbar im Workspace (`.github/instructions/`) oder User Profile

Dateiformat:

```
---
description: "Beschreibung der Instructions"
name: "Anzeigename"
applyTo: "**/*.ts"  # Glob-Pattern
---

## Anweisungen hier als Markdown
```

Speicherorte:

Ort	Pfad	Verfügbarkeit
Workspace	`.github/instructions/*.instructions.md`	Nur dieser Workspace
User Profile	VS Code Profile-Ordner	Alle Workspaces

`AGENTS.md`

Eigenschaften:

- Für Workspaces mit mehreren KI-Agents
- Datei im Workspace-Root
- Gilt für alle Chat-Requests

Aktivierung:

```
{
  "chat.useAgentsMdFile": true
}
```

1.3.3. Sprachabhängige Instructions

Mit dem `applyTo` Glob-Pattern können Sie Instructions für spezifische Programmiersprachen definieren:

Python

Datei: `.github/instructions/python.instructions.md`

```
---
applyTo: "**/*.py"
description: "Python Coding Standards"
---

## Python-Projektstandards

### Style Guide
```

```
- Befolge PEP 8
- Verwende Type Hints für alle Funktionen
- Docstrings im Google-Format

### Imports
- Gruppiere: Standard Library → Third Party → Local
- Absolute Imports bevorzugen

### Beispiel
```python
def calculate_total(items: list[Item]) -> Decimal:
    """Berechnet die Gesamtsumme aller Items.

    Args:
        items: Liste der zu berechnenden Items

    Returns:
        Gesamtsumme als Decimal
    """
    return sum(item.price for item in items)
```
```

```
#### TypeScript/React

**Datei:** `.github/instructions/typescript.instructions.md`

```text
---
applyTo: "**/*.ts,**/*.tsx"
description: "TypeScript/React Standards"
---

## TypeScript & React Guidelines

### TypeScript
- Verwende `interface` für Objektstrukturen
- Bevorzuge `const` und `readonly`
- Nutze Optional Chaining (`?.`) und Nullish Coalescing (`??`)

### React
- Funktionale Komponenten mit Hooks
- React.FC für Komponenten mit children
- Komponenten klein und fokussiert halten

### Naming
- PascalCase für Komponenten und Interfaces
- camelCase für Variablen und Funktionen
- UPPER_CASE für Konstanten
```
```

Dart/Flutter

Datei: `.github/instructions/dart.instructions.md`

```
---
applyTo: "**/*.dart"
description: "Dart/Flutter Conventions"
---

## Dart/Flutter Standards

### Code Style
- Befolge Effective Dart Guidelines
- Verwende `final` wo möglich
- Bevorzuge `const` Konstruktoren

### Flutter Widgets
- Stateless vor Stateful bevorzugen
- Widgets klein und wiederverwendbar halten
- BuildContext nicht in async Lücken verwenden

### Architektur
- Riverpod für State Management
- Clean Architecture Schichten einhalten
```

1.3.4. Verzeichnis-basierte Instructions

Neben Dateitypen können Sie auch Verzeichnisse als Kriterium verwenden:

Test-Verzeichnis

Datei: `.github/instructions/testing.instructions.md`

```
---
applyTo: "tests/**,test/**,*_test.dart,*.test.ts"
description: "Testing Guidelines"
---

## Test-Anweisungen

### Struktur
- Ein Test pro Datei für eine Klasse/Funktion
- Gruppierung mit describe/group
- Sprechende Testnamen

### Pattern
- Arrange: Setup der Testdaten
- Act: Ausführung der zu testenden Funktion
- Assert: Überprüfung des Ergebnisses

### Mocking
```

```
- Mocke externe Abhängigkeiten
- Verwende Fakes für komplexe Objekte
- Keine echten API-Calls in Unit Tests
```

Dokumentation

Datei: `.github/instructions/docs.instructions.md`

```
---
applyTo: "docs/**/*.md"
description: "Documentation Guidelines"
---

## Dokumentations-Richtlinien

### Sprache
- Präsens verwenden ("ist" statt "war")
- Aktive Sprache bevorzugen
- Direkte Ansprache ("Sie/Du")

### Format
- Überschriften zur Strukturierung
- Codeblöcke mit Syntax-Highlighting
- Links zu verwandten Ressourcen

### Inhalt
- Klare und präzise Formulierungen
- Praxisbeispiele wo sinnvoll
- Versionierung bei API-Dokumentation
```

Domain Layer (DDD)

Datei: `.github/instructions/domain.instructions.md`

```
---
applyTo: "src/domain/**,lib/domain/**"
description: "Domain-Driven Design Patterns"
---

## Domain Layer Guidelines

### Entities
- Eindeutige Identität über ID
- Geschäftslogik in der Entity
- Keine Framework-Abhängigkeiten

### Value Objects
- Immutabel
- Gleichheit über Werte, nicht Identität
```

```
- Validierung im Konstruktor

### Aggregate Roots
- Einziger Einstiegspunkt zu Aggregat
- Invarianten schützen
- Transaktionsgrenzen beachten
```

1.3.5. Referenzen zwischen Instruction-Dateien

Sie können Instruction-Dateien modular aufbauen und aufeinander referenzieren:

Datei: `.github/instructions/general.instructions.md`

```
---
applyTo: "**"
description: "General Coding Standards"
---

## Allgemeine Coding-Standards

### Naming Conventions
- PascalCase für Klassen und Interfaces
- camelCase für Variablen und Methoden
- Keine Abkürzungen in Namen
```

Datei: `.github/instructions/typescript.instructions.md`

```
---
applyTo: "**/*.ts"
description: "TypeScript specific"
---

Wende die [allgemeinen Coding-Standards](./general.instructions.md) an.

## Zusätzliche TypeScript-Regeln

### Typisierung
- Keine `any` Types
- Explizite Return-Types für öffentliche Funktionen
```

1.3.6. Tool-Referenzen in Instructions

Sie können in Instructions auf verfügbare Tools verweisen:

```
## Code-Analyse Instructions

Wenn du Code analysieren sollst, nutze #tool:search um den
Codebase-Kontext zu verstehen und #tool:githubRepo für
Repository-Informationen.
```

Verfügbare Tools:

- `#tool:search` – Codebase-Suche
- `#tool:fetch` – URL-Inhalte abrufen
- `#tool:githubRepo` – GitHub Repository Informationen
- `#tool:usages` – Symbol-Verwendungen finden

1.3.7. Best Practices

Do's ✅

1. **Spezifisch sein:** Konkrete Regeln statt vager Anweisungen
2. **Beispiele geben:** Code-Snippets zeigen gewünschtes Verhalten
3. **Modular aufbauen:** Kleine, fokussierte Instruction-Dateien
4. **Konsistent bleiben:** Einheitliche Terminologie verwenden

Don'ts ❌

1. **Zu allgemein:** "Schreibe guten Code" hilft nicht
2. **Widersprüchlich:** Instructions sollten sich nicht widersprechen
3. **Zu lang:** Übermäßig lange Instructions verwässern die Wirkung
4. **Veraltete Techniken:** Instructions aktuell halten

1.3.8. Quellen und Referenzen

Quelle	Beschreibung	Link
VS Code Custom Instructions	Offizielle Dokumentation	code.visualstudio.com/.../custom-instructions
GitHub Copilot Instructions	GitHub Dokumentation	docs.github.com/.../copilot-instructions
Awesome Copilot Instructions	Community-Beispiele	github.com/github/awesome-copilot
Glob Pattern Referenz	Pattern-Syntax	code.visualstudio.com/.../glob-patterns

---

1.4. Custom Agents

1.4.1. Was sind Custom Agents?

Definition: Custom Agents sind spezialisierte KI-Konfigurationen mit vordefinierten Anweisungen und Tools für spezifische Aufgaben. Sie ermöglichen schnelles Umschalten zwischen verschiedenen Arbeitsmodi.

Unterschied zu Custom Instructions

Custom Instructions	Custom Agents
Globale Coding-Standards	Aufgabenspezifische Konfiguration
Immer aktiv	Per Dropdown auswählbar
Nur Anweisungen	Anweisungen + Tool-Auswahl + Handoffs

> 💡 **Merke:** Agents wurden früher "Chat Modes" genannt. VS Code erkennt noch `.chatmode.md` Dateien, empfohlen ist jedoch `.agent.md`.

1.4.2. Agent-Dateistruktur

Speicherorte

```
Workspace:     .github/agents/*.agent.md
User Profile:  <VS Code Profile-Ordner>/*.agent.md
```

Dateiformat

```yaml
---
description: Kurzbeschreibung für Dropdown
name: Anzeigename
tools: ['tool1', 'tool2', 'my-mcp-server/*']
model: Claude Sonnet 4
mcp-servers:
  - my-mcp-server
handoffs:
  - label: Button-Text
    agent: ziel-agent
    prompt: Vorausgefüllter Prompt
    send: false
---

## Agent Instructions

Hier stehen die Markdown-Anweisungen für den Agent.
```

Header-Eigenschaften

Eigenschaft	Beschreibung
`description`	Kurzbeschreibung im Agent-Dropdown
`name`	Anzeigename des Agents
`tools`	Liste verfügbarer Tools (inkl. MCP-Server)
`model`	LLM-Modell (optional)
`mcp-servers`	Liste der MCP-Server für diesen Agent
`handoffs`	Übergaben zu anderen Agents

1.4.3. Tools: MCP-Server und VS Code Extensions

Agents können auf verschiedene Tools zugreifen, um ihre Fähigkeiten zu erweitern.

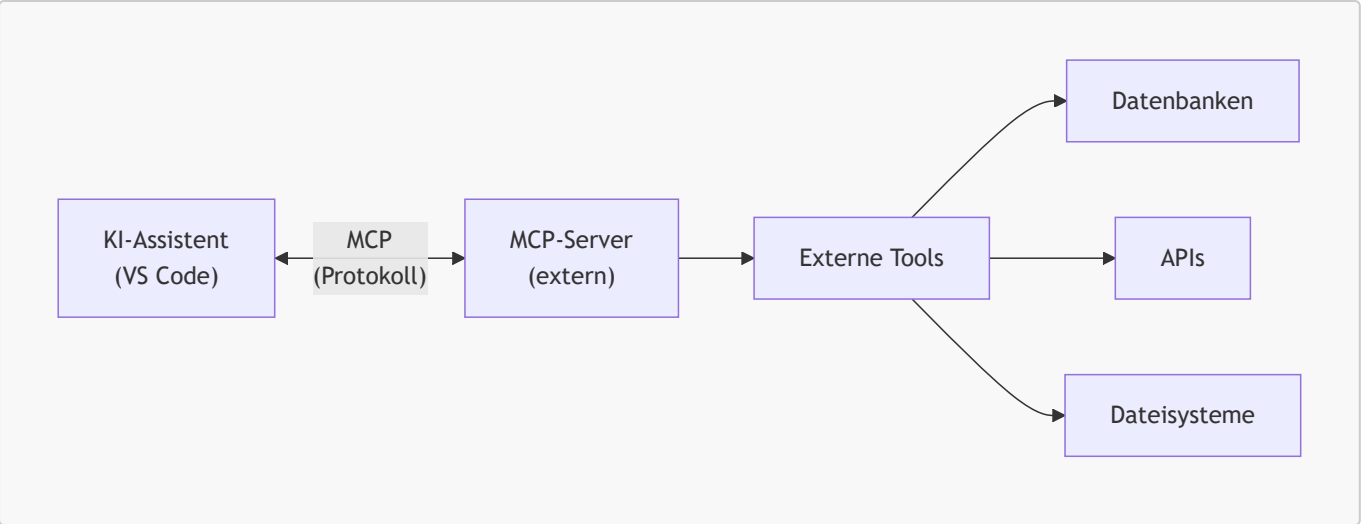
Built-in VS Code Tools

```
tools: ['search', 'fetch', 'githubRepo', 'usages', 'runTerminalCommand']
```

Tool	Beschreibung
`search`	Codebase durchsuchen
`fetch`	URLs abrufen
`githubRepo`	Repository-Informationen
`usages`	Symbol-Verwendungen finden
`runTerminalCommand`	Terminal-Befehle ausführen

Was ist das Model Context Protocol (MCP)?

Definition: MCP (Model Context Protocol) ist ein offenes Protokoll, das KI-Assistenten ermöglicht, sicher mit externen Datenquellen und Tools zu kommunizieren.



Warum MCP-Server verwenden?

Ohne MCP	Mit MCP
KI kann nur auf lokale Dateien zugreifen	Zugriff auf externe Datenquellen
Begrenzte Tool-Auswahl	Unbegrenzt erweiterbar
Jede Integration manuell	Standardisiertes Protokoll
Keine domänenspezifischen Tools	Spezialisierte Server pro Domäne

MCP-Server Beispiele

MCP-Server	Zweck
`dart-mcp-server`	Dart/Flutter Entwicklung (Analyze, Format, Tests)
`github-mcp-server`	GitHub Issues, PRs, Actions
`database-mcp`	SQL-Datenbankabfragen
`web-search-mcp`	Web-Recherche
`filesystem-mcp`	Erweiterter Dateizugriff

MCP-Server Konfiguration

1. In `settings.json` (VS Code):

```
{
  "mcp": {
    "servers": {
      "dart-mcp-server": {
        "command": "dart",
        "args": ["run", "dart_mcp_server"],
        "cwd": "${workspaceFolder}"
      },
```

```
      "github": {
        "command": "npx",
        "args": ["-y", "@modelcontextprotocol/server-github"],
        "env": {
          "GITHUB_TOKEN": "${env:GITHUB_TOKEN}"
        }
      }
    }
  }
}
```

2. In `.vscode/mcp.json` (Workspace-spezifisch):

```
{
  "servers": {
    "project-specific-server": {
      "command": "node",
      "args": ["./scripts/mcp-server.js"]
    }
  }
}
```

MCP-Tools in Agents verwenden

Im Agent-Header:

```
---
description: Flutter Development Agent
name: Flutter Dev
tools: ['search', 'dart-mcp-server/*']  # Alle Tools des Servers
mcp-servers:
  - dart-mcp-server
---
```

Im Agent-Body:

```
## Flutter Development Instructions

Verwende #tool:dart-mcp-server/analyze_files für Code-Analyse.
Führe Tests aus mit #tool:dart-mcp-server/run_tests.
Formatiere Code mit #tool:dart-mcp-server/dart_format.
```

Verfügbare Tools eines MCP-Servers anzeigen

Um zu sehen, welche Tools ein MCP-Server bereitstellt:

1. MCP-Server in `settings.json` konfigurieren
2. In VS Code Chat: "Welche Tools hat der dart-mcp-server?"
3. Oder: Command Palette → "MCP: List Tools"

Beispiel: Dart MCP-Server Tools

Tool	Funktion
`analyze_files`	Dart Analyzer ausführen
`dart_format`	Code formatieren
`dart_fix`	Automatische Fixes anwenden
`run_tests`	Tests ausführen
`pub`	Pub-Befehle (get, add, upgrade)
`hot_reload`	Flutter Hot Reload
`launch_app`	Flutter App starten

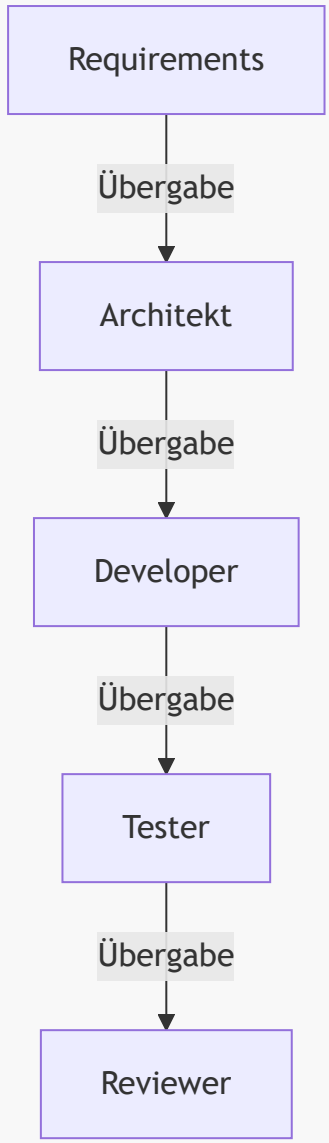
> 💡 **Tipp:** MCP-Server können auch eigene **Resources** bereitstellen, wie z.B. aktuelle Fehlerlisten oder Widget-Trees.

MCP-Server Quellen und Referenzen

Quelle	Beschreibung	Link
MCP Spezifikation	Offizielle Protokoll-Dokumentation	modelcontextprotocol.io
MCP GitHub	Offizielle Repositories und SDKs	github.com/modelcontextprotocol
Awesome MCP Servers	Kuratierte Liste von Community-Servern	github.com/punkpeye/awesome-mcp-servers
VS Code MCP Docs	VS Code Integration Dokumentation	code.visualstudio.com/.../mcp
MCP .so	Find awesom MCP Servers and Clients	mcp.so
Smithery	MCP-Server Registry und Discovery	smithery.ai

1.4.4. Sequentielle Agent-Verkettung (Handoffs)

Handoffs ermöglichen geführte Workflows zwischen Agents. Sie sollten verstanden werden, **bevor** die einzelnen Agent-Rollen betrachtet werden, da jeder Agent Handoffs zu anderen Agents definiert.



Handoff-Konfiguration

```yaml
handoffs:
  - label: "Start Implementation"    # Button-Text
    agent: developer                 # Ziel-Agent Dateiname (ohne .agent.md)
    prompt: "Implementiere den Plan" # Vorausgefüllter Prompt
    send: false                      # false = User kann editieren vor Absenden
```

Wie Handoffs funktionieren

1. Agent bearbeitet eine Anfrage und generiert eine Antwort
2. Am Ende der Antwort erscheint ein **Handoff-Button**
3. Klick auf Button wechselt zum Ziel-Agent mit vorausgefülltem Prompt
4. Bei `send: true` wird der Prompt automatisch abgesendet

Beispiel-Workflow

Schritt	Agent	Aktion	Handoff-Button
1	Requirements	User Stories erstellen	"→ Zur Architektur"
2	Architekt	Design erstellen	"→ Implementierung"
3	Developer	Code schreiben	"→ Tests erstellen"
4	Tester	Tests schreiben	–

1.4.5. Agent-Rollen für Software-Entwicklung

> 💡 **Tipp:** Für die Verwendung von Dokumenten-Templates in Agents siehe Kapitel 5: Best Practices.

Requirements Engineer Agent

Datei: `.github/agents/requirements.agent.md`

```
---
description: Erstellt User Stories und Akzeptanzkriterien
name: Requirements Engineer
tools: ['search', 'fetch', 'githubRepo']
handoffs:
  - label: Zur Architektur
    agent: architect
    prompt: Erstelle basierend auf diesen Requirements eine Architektur.
    send: false
---

## Requirements Engineering Instructions

Du bist ein Requirements Engineer. Deine Aufgabe ist es:

### User Stories erstellen
- Format: "Als [Rolle] möchte ich [Funktion], damit [Nutzen]"
- Klare, testbare Akzeptanzkriterien definieren
- INVEST-Kriterien einhalten (Independent, Negotiable, Valuable, Estimable, Small,
Testable)

### Analyse
- Stakeholder-Anforderungen erfassen
- Abhängigkeiten identifizieren
- Priorisierung nach MoSCoW vorschlagen

### Deliverables
- User Story mit Akzeptanzkriterien
- Story Map für Feature-Übersicht
- Definition of Done
```

SW-Architekt Agent

Datei: `.github/agents/architect.agent.md`

```
---
description: Erstellt Software-Architektur und Domain-Modelle
name: SW-Architekt
tools: ['search', 'usages', 'githubRepo']
handoffs:
  - label: Implementierung starten
    agent: developer
    prompt: Implementiere die Architektur gemäß dem obigen Plan.
    send: false
---

## Software Architecture Instructions

Du bist ein Software-Architekt. Folge diesen Prinzipien:

### Architektur-Patterns
- Clean Architecture mit klarer Schichtentrennung
- Domain-Driven Design für komplexe Domänen
- SOLID-Prinzipien einhalten

### Domain Modeling
- Bounded Contexts identifizieren
- Aggregates und Entities definieren
- Value Objects für unveränderliche Konzepte

### Dokumentation
- Architektur-Entscheidungen dokumentieren (ADRs)
- Komponenten-Diagramme erstellen
- API-Contracts definieren
```

Frontend-Developer Agent (CDD)

Datei: `.github/agents/frontend.agent.md`

```
---
description: UI-Entwicklung mit Component-Driven Development
name: Frontend Developer
tools: ['search', 'fetch', 'usages']
handoffs:
  - label: Tests erstellen
    agent: tester
    prompt: Erstelle Tests für die implementierten Komponenten.
    send: false
---

## Frontend Development Instructions
```

```
Du bist ein Frontend-Entwickler mit Fokus auf Component-Driven Development (CDD).

### CDD-Prinzipien
- Bottom-Up: Atoms → Molecules → Organisms → Templates → Pages
- Komponenten isoliert entwickeln und testen
- Storybook für Komponenten-Dokumentation

### Komponenten-Design
- Single Responsibility: Eine Komponente, eine Aufgabe
- Props für Konfiguration, Events für Kommunikation
- Accessibility (a11y) von Anfang an

### Styling
- Design Tokens für konsistente Werte
- Responsive Design mit Mobile-First
- CSS Modules oder Styled Components
```

Backend-Developer Agent

Datei: `.github/agents/backend.agent.md`

```
---
description: Backend-Entwicklung mit Clean Architecture
name: Backend Developer
tools: ['search', 'usages', 'githubRepo']
---

## Backend Development Instructions

Du bist ein Backend-Entwickler. Folge diesen Prinzipien:

### API-Design
- RESTful Conventions einhalten
- OpenAPI/Swagger für Dokumentation
- Versionierung von Anfang an

### Business Logic
- Use Cases im Application Layer
- Domain-Logik in Entities und Value Objects
- Services für komplexe Operationen

### Persistence
- Repository Pattern für Datenzugriff
- Database Migrations verwenden
- Optimistic Locking bei Concurrency
```

Test-Engineer Agent

Datei: `.github/agents/tester.agent.md`

```
---
description: Erstellt Unit-, Integration- und E2E-Tests
name: Test Engineer
tools: ['search', 'usages', 'runTerminalCommand']
---

## Testing Instructions

Du bist ein Test-Engineer. Erstelle umfassende Tests:

### Unit Tests
- Arrange-Act-Assert Pattern
- Eine Assertion pro Test (idealerweise)
- Mocking für externe Abhängigkeiten

### Integration Tests
- Reale Datenbankverbindungen testen
- API-Endpoints end-to-end
- Keine Mocks für interne Komponenten

### Test Coverage
- Kritische Pfade 100% abdecken
- Edge Cases berücksichtigen
- Error-Handling testen
```

1.4.6. Antigravity Agent-Modi

Google Antigravity verwendet ein strukturiertes Modus-System:

Die drei Modi

Modus	Zweck	Typische Aktionen
PLANNING	Recherche und Design	Codebase analysieren, Plan erstellen
EXECUTION	Implementierung	Code schreiben, Dateien ändern
VERIFICATION	Validierung	Tests ausführen, Review erstellen

Task Boundaries

```
task_boundary(
  Mode: "EXECUTION",
  TaskName: "Implementing User Authentication",
  TaskStatus: "Creating login endpoint",
```

```
  TaskSummary: "Completed user model and repository."
)
```

Artifacts

Antigravity erstellt automatisch strukturierte Dokumente:

- `task.md` – Fortschrittsverfolgung
- `implementation_plan.md` – Technischer Plan
- `walkthrough.md` – Dokumentation der Änderungen

1.4.7. Agent erstellen (Schritt-für-Schritt)

In VS Code

1. **Agent-Dropdown öffnen** → "Configure Custom Agents"
2. **"Create new custom agent"** auswählen
3. **Speicherort wählen:**
   - Workspace: `.github/agents/`
   - User Profile: Überall verfügbar
4. **Dateiname eingeben** (wird zum Agent-Namen)
5. **YAML Header ausfüllen** (description, tools)
6. **Instructions schreiben**

Migration von Chat Modes

Alte `.chatmode.md` Dateien können migriert werden:

- Quick Fix nutzen zum Umbenennen
- Neue Eigenschaften (`handoffs`, `mcp-servers`) ergänzen

1.4.8. Quellen und Referenzen

Quelle	Beschreibung	Link
VS Code Custom Agents	Offizielle Dokumentation	code.visualstudio.com/.../custom-agents
MCP Spezifikation	Model Context Protocol	modelcontextprotocol.io
MCP GitHub	Offizielle Repositories	github.com/modelcontextprotocol
Awesome MCP Servers	Community-Server-Liste	github.com/punkpeye/awesome-mcp-servers
MCP .so	Server und Client Discovery	mcp.so
Smithery	MCP Registry	smithery.ai
Awesome Copilot	Agent-Beispiele	github.com/github/awesome-copilot

---

1.5. Custom Prompts & Workflows

Dieses Kapitel behandelt wiederholbare Prompt-Vorlagen für häufige Aufgaben. VS Code verwendet **Custom Prompts**, während Google Antigravity **Workflows** nutzt.

1.5.1. Was sind Custom Prompts / Workflows?

Definition: Custom Prompts (VS Code) und Workflows (Antigravity) sind vordefinierte Anweisungsvorlagen für wiederkehrende Aufgaben. Sie ermöglichen schnelles Ausführen von Standardoperationen.

Unterschied zu Agents

Custom Agents	Custom Prompts / Workflows
Komplexe Rollendefinition	Einfache Aufgabenvorlage
Können Tools einbinden	Nur Prompt-Text
Handoffs zu anderen Agents	Keine Verkettung
Dauerhafte Konfiguration	Einmalige Ausführung

> 💡 **Merke:** Prompts/Workflows sind für **einfache, wiederholbare Aufgaben**. Für komplexe Szenarien mit Tool-Nutzung verwende Agents.

1.5.2. VS Code Custom Prompts

Speicherorte

```
Workspace:     .github/prompts/*.prompt.md
User Profile:  <VS Code Profile-Ordner>/*.prompt.md
```

Dateiformat

```
---
description: Kurzbeschreibung für Dropdown
mode: agent | ask | edit
---

## Prompt Instructions

Hier stehen die Markdown-Anweisungen für den Prompt.
Variablen können mit ${variable} eingefügt werden.
```

Header-Eigenschaften

Eigenschaft	Beschreibung
`description`	Kurzbeschreibung im Prompt-Dropdown
`mode`	Ausführungsmodus: `agent` (Standard), `ask`, oder `edit`

Modi erklärt

Modus	Beschreibung	Anwendung
`agent`	Agent mit vollem Tool-Zugriff	Komplexe Aufgaben, Dateierstellung
`ask`	Nur Antwort, keine Dateiänderungen	Erklärungen, Recherche
`edit`	Inline-Editor für Änderungen	Code-Refactoring, Quick Fixes

---

Beispiel: Unit Test Generator

Datei: `.github/prompts/generate-tests.prompt.md`

```
---
description: Generiert Unit Tests für die aktuelle Datei
mode: agent
---

## Unit Test Generator

Erstelle umfassende Unit Tests für den aktuellen Code.

### Anforderungen

- Verwende das Arrange-Act-Assert Pattern
- Teste alle öffentlichen Methoden
- Berücksichtige Edge Cases und Fehlerszenarien
- Mocke externe Abhängigkeiten
- Ziel: 80% Code Coverage

### Ausgabeformat

Erstelle eine Testdatei im passenden Test-Ordner mit dem Suffix `_test` oder
`.test`.
```

Beispiel: Code Review Prompt

Datei: `.github/prompts/code-review.prompt.md`

```
---
description: Führt ein Code Review durch
```

```
mode: ask
---

## Code Review

Analysiere den ausgewählten Code und gib konstruktives Feedback:

### Prüfpunkte

1. **Lesbarkeit**: Sind Namen aussagekräftig? Ist der Code verständlich?
2. **SOLID-Prinzipien**: Werden sie eingehalten?
3. **Error Handling**: Sind Fehler korrekt behandelt?
4. **Performance**: Gibt es offensichtliche Optimierungsmöglichkeiten?
5. **Tests**: Ist der Code testbar? Fehlen Tests?

### Ausgabeformat

Gib Feedback als Liste mit Priorität (Kritisch, Wichtig, Optional).
```

Beispiel: Dokumentation Generator

Datei: `.github/prompts/generate-docs.prompt.md`

```
---
description: Generiert Dokumentation für Code
mode: edit
---

## Dokumentation Generator

Ergänze fehlende Dokumentation im aktuellen Code:

- Füge JSDoc/Dartdoc/Docstrings hinzu
- Beschreibe Parameter und Rückgabewerte
- Dokumentiere Exceptions/Errors
- Füge Beispiele bei komplexen Funktionen hinzu

Halte die Dokumentation prägnant aber vollständig.
```

Variablen in Prompts

Custom Prompts können Variablen nutzen:

Variable	Beschreibung
`${file}`	Aktueller Dateiname
`${selection}`	Ausgewählter Text

Variable	Beschreibung
`${input:Name}`	Benutzereingabe mit Label
`${clipboard}`	Inhalt der Zwischenablage

Beispiel mit Variablen:

```
---
description: Erklärt Code in gewählter Sprache
mode: ask
---

Erkläre den folgenden Code auf ${input:Sprache}:

${selection}
```

1.5.3. Antigravity Workflows

Google Antigravity verwendet ein ähnliches Konzept namens **Workflows**.

Speicherort

```
Workspace: .agent/workflows/*.md
```

Dateiformat

```
---
description: Workflow-Beschreibung
---

## Workflow Name

Schrittweise Anweisungen für den Workflow.
Kann auf andere Dateien und Ressourcen verweisen.
```

> 💡 **Hinweis:** Antigravity Workflows werden mit `/workflow-name` in der Chat-Eingabe aufgerufen.

Beispiel: Feature Implementation Workflow

Datei: `.agent/workflows/implement-feature.md`

```
---
description: Implementiert ein neues Feature nach Best Practices
---

## Feature Implementation Workflow

### Schritt 1: Analyse

- Analysiere die bestehende Codebase
- Identifiziere betroffene Komponenten
- Prüfe auf ähnliche bestehende Implementierungen

### Schritt 2: Planung

- Erstelle einen Implementation Plan
- Definiere die benötigten Änderungen pro Datei
- Identifiziere potenzielle Risiken

### Schritt 3: Implementierung

- Implementiere die Änderungen schrittweise
- Folge den Projekt-Konventionen
- Schreibe Tests parallel zur Implementierung

### Schritt 4: Verifizierung

- Führe alle Tests aus
- Prüfe auf Lint-Fehler
- Erstelle einen Walkthrough der Änderungen
```

Beispiel: Bug Fix Workflow

Datei: `.agent/workflows/fix-bug.md`

```
---
description: Systematisches Debugging und Bugfix
---

## Bug Fix Workflow

### Schritt 1: Reproduktion

- Verstehe den gemeldeten Fehler
- Reproduziere das Problem lokal
- Identifiziere die Fehlerbedingungen

### Schritt 2: Analyse

- Finde die Ursache (Root Cause Analysis)
```

```
- Prüfe verwandte Code-Bereiche
- Dokumentiere deine Erkenntnisse

### Schritt 3: Fix

- Implementiere die Korrektur
- Halte die Änderung minimal und fokussiert
- Vermeide Seiteneffekte

### Schritt 4: Test

- Schreibe einen Test, der den Bug reproduziert
- Verifiziere, dass der Test jetzt besteht
- Führe Regression-Tests durch
```

Beispiel: Code Review Workflow

Datei: `.agent/workflows/review-code.md`

```
---
description: Systematisches Code Review
---

## Code Review Workflow

### Prüfliste

#### 1. Funktionalität
- [ ] Erfüllt der Code die Anforderungen?
- [ ] Sind Edge Cases berücksichtigt?
- [ ] Ist Error Handling vorhanden?

#### 2. Code-Qualität
- [ ] Ist der Code lesbar und verständlich?
- [ ] Werden Naming Conventions eingehalten?
- [ ] Gibt es Code-Duplikationen?

#### 3. Architektur
- [ ] Passt der Code zur bestehenden Architektur?
- [ ] Werden SOLID-Prinzipien eingehalten?
- [ ] Sind Abhängigkeiten korrekt?

#### 4. Tests
- [ ] Sind ausreichend Tests vorhanden?
- [ ] Decken Tests die kritischen Pfade ab?
- [ ] Sind Tests lesbar und wartbar?

### Ausgabe

Erstelle einen Review-Bericht mit:
```

```
- Zusammenfassung
- Kritische Issues
- Verbesserungsvorschläge
- Positive Aspekte
```

1.5.4. Turbo-Annotation für automatische Ausführung

Antigravity Workflows unterstützen eine spezielle Annotation für automatische Befehlsausführung:

```
### Schritt 1: Dependencies installieren

// turbo
Führe `npm install` aus.

### Schritt 2: Tests ausführen

// turbo
Führe `npm test` aus.

### Schritt 3: Build erstellen

Führe `npm run build` aus.  // Hier wird nachgefragt (kein turbo)
```

Annotation	Verhalten
`// turbo`	Einzelner Schritt wird automatisch ausgeführt
`// turbo-all`	Alle Schritte im Workflow werden automatisch ausgeführt

> ⚠️ **Achtung:** Verwende `turbo` nur für sichere, nicht-destruktive Befehle!

1.5.5. Custom Prompts/Workflows erstellen

In VS Code

1. **Command Palette öffnen** → "Configure Custom Prompts"
2. **"Create new prompt"** auswählen
3. **Speicherort wählen:**
   - Workspace: `.github/prompts/`
   - User Profile: Überall verfügbar
4. **Dateiname eingeben** (z.B. `generate-tests.prompt.md`)
5. **YAML Header mit description und mode ausfüllen**
6. **Prompt-Anweisungen schreiben**

In Antigravity

1. **Ordner erstellen:** `.agent/workflows/`

2. **Neue Datei anlegen** (z.B. `fix-bug.md`)
3. **YAML Header mit description ausfüllen**
4. **Workflow-Schritte dokumentieren**
5. **Aufrufen mit** `/fix-bug` im Chat

1.5.6. Best Practices

Prompt-Design

Do	Don't
Klare, spezifische Anweisungen	Vage, mehrdeutige Beschreibungen
Strukturierte Ausgabeformate	Unstrukturierte Freitextausgabe
Beispiele für komplexe Aufgaben	Annahmen über Vorwissen
Variablen für Flexibilität	Hardcodierte Werte

Workflow-Organisation

```
.github/prompts/              # VS Code
├── development/
│   ├── generate-tests.prompt.md
│   └── refactor.prompt.md
├── documentation/
│   ├── generate-docs.prompt.md
│   └── readme-update.prompt.md
└── review/
    ├── code-review.prompt.md
    └── security-review.prompt.md

.agent/workflows/             # Antigravity
├── implement-feature.md
├── fix-bug.md
└── review-code.md
```

> 💡 **Tipp:** Für die Verwendung von Dokumenten-Templates in Prompts/Workflows siehe Kapitel 5: Best Practices.

1.5.7. Quellen und Referenzen

Quelle	Beschreibung	Link
VS Code Reusable Prompts	Offizielle Dokumentation	code.visualstudio.com/.../prompt-crafting
Prompt Engineering Guide	Best Practices für Prompts	platform.openai.com/docs

---

1.6. Best Practices & Erweiterungen

Dieses Kapitel behandelt bewährte Praktiken und funktionale Erweiterungen, die in Kombination mit `Custom Instructions`, `Agents` und `Workflows` eingesetzt werden können.

>  **Merksatz:** Standardisiere Quellen und Templates, frage vor dem Erstellen nach, und halte Kontexte schlank – so bleiben KI-gestützte Prozesse effizient und verlässlich.

1.6.1. Dokumenten-Templates

Warum Templates verwenden?

Templates für Dokumente bieten konsistente Strukturen und reduzieren den Aufwand für den Agent bei der Dokumentenerstellung.

Vorteil	Beschreibung
Konsistenz	Einheitliche Dokumentenstruktur im Projekt
Effizienz	Agent muss Struktur nicht jedes Mal neu erstellen
Qualität	Vordefinierte Abschnitte verhindern Auslassungen
Projektstandards	Templates spiegeln Team-Konventionen wider

Template-Speicherorte

```
.github/
├── agents/
│   ├── requirements.agent.md
│   └── architect.agent.md
└── templates/                    # Empfohlener Ordner
    ├── user-story.md
    ├── architecture-decision-record.md
    ├── test-plan.md
    └── review-checklist.md
```

> 💡 **Empfehlung:** Speichere Templates in `.github/templates/` oder `docs/templates/` als Default-Ordner.

Templates im Agent referenzieren

Im Agent-Body auf Templates verweisen:

```
## Requirements Engineering Instructions

### Dokumentenerstellung

Verwende die folgenden Templates aus `.github/templates/`:

- **User Stories:** `user-story.md`
- **Feature-Spezifikation:** `feature-spec.md`
- **Story Map:** `story-map.md`

**WICHTIG:** Bevor du ein Dokument erstellst:
1. Frage den Benutzer, ob das Dokument erstellt werden soll
```

```
2. Schlage den Standard-Speicherort vor (z.B. `docs/requirements/`)
3. Biete an, einen alternativen Speicherort zu verwenden
4. Erstelle das Dokument erst nach Bestätigung
```

Templates in Workflows verwenden

In VS Code Custom Prompts:

```
---
description: Erstellt eine User Story nach Template
---

Verwende das Template aus `.github/templates/user-story.md` um eine User Story zu
erstellen.

**Vor der Erstellung:**
1. Frage ob das Dokument erstellt werden soll
2. Schlage `docs/requirements/user-stories/` als Speicherort vor
3. Biete Alternativen an

**Nach Bestätigung:**
- Lade das Template
- Fülle die Platzhalter aus
- Speichere am gewählten Ort
```

In Antigravity Workflows:

```
---
description: User Story erstellen
---

### Workflow-Schritte

1. Analysiere die Anforderung des Benutzers
2. Lade Template: `.github/templates/user-story.md`
3. Frage den Benutzer:
   - "Soll ich eine User Story erstellen?"
   - "Standard-Speicherort: `docs/requirements/user-stories/` – OK oder anderer
Ordner?"
4. Warte auf Bestätigung
5. Erstelle Dokument basierend auf Template
```

Best Practice: Benutzer-Interaktion vor Dokumentenerstellung

Agents und Workflows sollten **aktiv nachfragen**, bevor sie Dokumente erstellen:

```
### Interaktionsregeln für Dokumentenerstellung

Wenn du ein Dokument erstellen sollst:

1. **Bestätigung einholen:**
   "Soll ich eine User Story für [Feature] erstellen?"

2. **Standard-Speicherort vorschlagen:**
   "Der Standard-Speicherort wäre `docs/requirements/user-stories/`.
   Möchtest du diesen verwenden oder einen anderen Ordner angeben?"

3. **Bei Ablehnung:** Alternativen anbieten oder auf Erstellung verzichten

4. **Bei Bestätigung:** Template aus `.github/templates/` laden und ausfüllen
```

Beispiel: Agent mit Template-Konfiguration

```
---
description: Erstellt User Stories nach Template
name: Requirements Engineer
tools: ['search', 'fetch']
handoffs:
  - label: Zur Architektur
    agent: architect
---

## Requirements Engineering

### Templates

Verwende diese Templates für Dokumentenerstellung:

| Dokument | Template | Default-Speicherort |
|----------|----------|---------------------|
| User Story | `.github/templates/user-story.md` | `docs/requirements/user-
stories/` |
| Epic | `.github/templates/epic.md` | `docs/requirements/epics/` |
| Story Map | `.github/templates/story-map.md` | `docs/requirements/` |

### Workflow für Dokumentenerstellung

1. Analysiere die Anforderung
2. **Frage den Benutzer:**
   - "Soll ich ein [Dokumenttyp]-Dokument erstellen?"
   - "Soll ich den Standard-Ordner `[pfad]` verwenden, oder möchtest du einen
anderen Speicherort angeben?"
3. Warte auf Bestätigung
4. Erstelle Dokument basierend auf Template
5. Fülle die Platzhalter im Template aus
```

> ⚠️ **Wichtig:** Erstelle niemals Dokumente ohne vorherige Rückfrage beim Benutzer!

Quellen

- Best Practices für Developer Experience und Templates (Projektinterne Konventionen)
- GitHub Docs – Repository Strukturierungen: https://docs.github.com

---

1.6.2. Kontext-Dateien & Referenzquellen

Agents können auf zusätzliche Informationsquellen zugreifen, wenn sie wissen **wo** sie relevante Informationen finden und **wann** sie diese nutzen sollten.

Arten von Kontextquellen

Quelle	Beschreibung	Anwendung
Projekt-Dokumentation	Lokale Markdown-Dateien	Architektur, Coding Standards
API-Dokumentation	URLs zu offiziellen Docs	Framework-spezifische Fragen
Beispiel-Repositories	GitHub-Links	Referenz-Implementierungen
Interne Wikis	Confluence, Notion etc.	Team-Konventionen

Kontext-Dateien im Projekt

Empfohlene Struktur

```
docs/
├── architecture/
│   ├── overview.md          # Architektur-Übersicht
│   ├── decisions/           # Architecture Decision Records
│   └── patterns.md          # Verwendete Design Patterns
├── conventions/
│   ├── coding-standards.md  # Code-Konventionen
│   ├── naming.md            # Naming Conventions
│   └── git-workflow.md      # Git Branching Strategy
├── domain/
│   ├── glossary.md          # Domänen-Glossar
│   └── bounded-contexts.md  # Domain-Driven Design Kontexte
└── api/
    ├── endpoints.md         # API-Dokumentation
    └── error-codes.md       # Fehlercode-Referenz
```

Quellen im Agent referenzieren

Quellenverzeichnis im Agent-Body:

```
## Development Agent Instructions

### Verfügbare Kontextquellen

Nutze folgende Quellen, wenn du zusätzliche Informationen benötigst:

| Thema | Quelle | Wann nutzen? |
|-------|--------|--------------|
| **Architektur** | `docs/architecture/overview.md` | Bei strukturellen Fragen
oder neuen Komponenten |
| **Coding Standards** | `docs/conventions/coding-standards.md` | Bei Code-
Formatierung oder Style-Fragen |
| **Domänen-Begriffe** | `docs/domain/glossary.md` | Bei unklaren Fachbegriffen |
| **API-Referenz** | `docs/api/endpoints.md` | Bei API-Implementierung oder
Integration |
| **Git-Workflow** | `docs/conventions/git-workflow.md` | Bei Branch-Erstellung
oder Commit-Messages |

### Anwendungsregel

Bevor du eine Frage beantwortest oder Code schreibst:
1. Prüfe, ob eine der obigen Quellen relevante Informationen enthält
2. Lies die Quelle, wenn du dir unsicher bist
3. Halte dich an die dort definierten Standards
```

Externe Links als Referenzen

Für Framework-/Technologie-spezifische Informationen:

```
## Flutter Development Agent

### Externe Referenzen

Nutze folgende offizielle Dokumentationen bei Bedarf:

| Thema | URL | Wann nutzen? |
|-------|-----|--------------|
| **Flutter Widgets** | https://api.flutter.dev/flutter/widgets/widgets-
library.html | Widget-Eigenschaften, Parameter |
| **Dart Language** | https://dart.dev/language | Sprachfeatures, Syntax |
| **Material Design** | https://m3.material.io/ | UI-Komponenten, Design
Guidelines |
| **Riverpod** | https://riverpod.dev/docs | State Management Patterns |
| **go_router** | https://pub.dev/packages/go_router | Navigation, Routing |

### Anwendungsregel
```

```
- Nutze **lokale Dokumentation** für projektspezifische Standards
- Nutze **externe Links** für Framework-/API-Details
- Bevorzuge **offizielle Dokumentation** vor Stack Overflow o.ä.
```

Kontext-Verzeichnis Beispiel (CONTEXT.md)

Erstelle eine zentrale Datei, die alle verfügbaren Quellen dokumentiert:

Datei: `docs/CONTEXT.md` oder `.github/CONTEXT.md`

```
## Projekt-Kontextverzeichnis

Diese Datei listet alle verfügbaren Informationsquellen für KI-Agenten.

### Lokale Dokumentation

| Datei | Beschreibung | Relevanz für |
|-------|--------------|--------------|
| `README.md` | Projekt-Übersicht, Setup | Neue Features, Onboarding |
| `docs/architecture/overview.md` | Architektur-Diagramme | Strukturelle
Änderungen |
| `docs/conventions/coding-standards.md` | Code-Konventionen | Jede Code-Änderung
|
| `docs/domain/glossary.md` | Fachbegriffe | Domain-spezifischer Code |
| `CHANGELOG.md` | Änderungshistorie | Version-Informationen |

### Externe Referenzen

| Thema | URL | Beschreibung |
|-------|-----|--------------|
| Flutter Docs | https://docs.flutter.dev | Offizielle Flutter-Dokumentation |
| Dart Docs | https://dart.dev/guides | Dart Sprachführer |
| Pub.dev | https://pub.dev | Package-Dokumentation |

### Wichtige Hinweise

- **Immer prüfen:** Bei Unsicherheit die relevante Quelle konsultieren
- **Priorität:** Lokale Konventionen > Offizielle Docs > Community-Lösungen
- **Aktualität:** Links können veralten, im Zweifel selbst suchen
```

Agent mit Kontextquellen-Verweis

Vollständiges Beispiel:

```
---
description: Backend-Entwicklung mit Clean Architecture
name: Backend Developer
```

```
tools: ['search', 'fetch', 'usages']
---

## Backend Development Instructions

Du bist ein Backend-Entwickler für dieses Projekt.

### Kontextquellen

**Lies zuerst diese Dateien, wenn du Fragen zu folgenden Themen hast:**

| Frage/Thema | Quelle |
|-------------|--------|
| Wie ist die Architektur aufgebaut? | `docs/architecture/overview.md` |
| Welche Coding Standards gelten? | `docs/conventions/coding-standards.md` |
| Was bedeutet ein Fachbegriff? | `docs/domain/glossary.md` |
| Wie sieht ein Use Case aus? | `src/application/use_cases/example_use_case.dart`
|
| Wie ist das Repository-Pattern? | `src/infrastructure/repositories/` |

**Externe Referenzen bei Bedarf:**

| Thema | URL |
|-------|-----|
| Dart Best Practices | https://dart.dev/effective-dart |
| Clean Architecture | https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-
architecture.html |

### Verhaltensregeln

1. **Vor jeder Implementierung:** Prüfe, ob relevante Konventionen existieren
2. **Bei Domain-Logik:** Konsultiere das Glossar für korrekte Begriffe
3. **Bei Architektur-Fragen:** Lies die Architektur-Dokumentation
4. **Bei Framework-Fragen:** Nutze die offiziellen Docs (fetch-Tool)
```

Best Practice: Kontext-Quellen effektiv nutzen

Do	Don't
Quellenverzeichnis zentral pflegen	Quellen überall verstreuen
Klare Beschreibung, wann Quelle nützlich ist	Nur Links ohne Kontext listen
Lokale und externe Quellen unterscheiden	Alles in einen Topf werfen
Prioritäten definieren (lokal vor extern)	Keine Reihenfolge angeben
Quellen aktuell halten	Veraltete Links stehen lassen

>  **Tipp:** Erstelle eine `CONTEXT.md` Datei im Projekt-Root oder in `.github/`, die alle verfügbaren Quellen mit kurzer Beschreibung listet. Referenziere diese Datei in deinen Agents.

Quellen

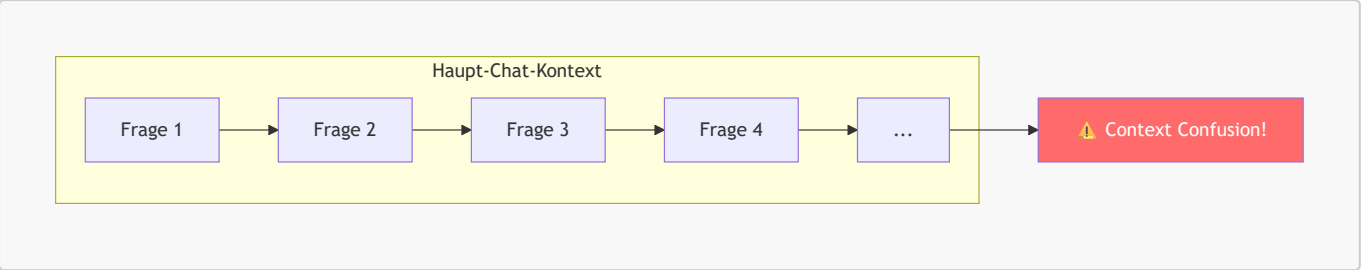
- Technische Dokumentation – Projektinterne `docs/` Strukturen
- Offizielle Framework-Dokumentationen (Flutter/Dart/Material): https://docs.flutter.dev, https://dart.dev, https://m3.material.io
- Clean Architecture: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

---

1.6.3. SubAgents für Kontextisolation (VS Code)

Das Problem: Context Confusion

Je länger ein Chat mit einem Agent dauert, desto mehr Kontext sammelt sich an. Das kann zu **Context Confusion** führen – der Agent verliert den Überblick oder vermischt Informationen aus verschiedenen Aufgaben.



Die Lösung: SubAgents

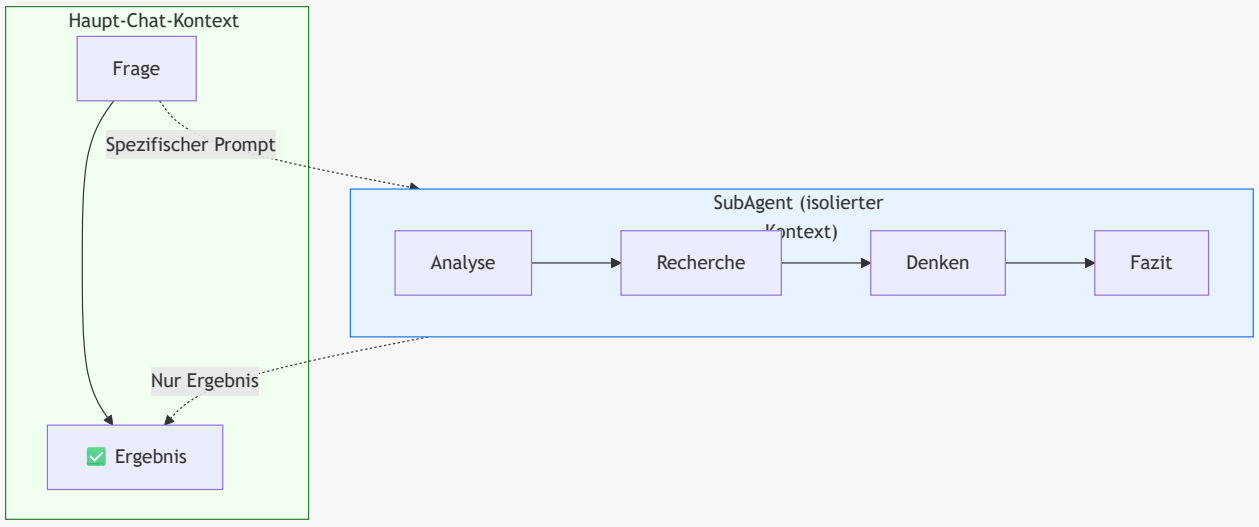
> 🔍 **Vertiefung:** SubAgents helfen, dedizierte Aufgaben kontextrein zu bearbeiten (z.B. isolierte Recherche), während der Haupt-Chat fokussiert bleibt. Nutze klare Übergaben und eindeutige Eingabe-/Ausgabe-Formate.

Quellen

- VS Code Agent- und Prompt-Workflows (projektinterne Guidelines)
- Prinzipien zu Kontextmanagement in LLM-gestützten Systemen (Community-Praxishinweise)

SubAgents sind isolierte Agent-Instanzen, die:

- **Eigenen Kontext** haben (unabhängig vom Haupt-Chat)
- **Nur die nötigen Informationen** erhalten
- **Nur das Ergebnis** an den Haupt-Chat zurückgeben
- **Ohne Unterbrechung** arbeiten (keine User-Feedback-Pausen)



SubAgent in VS Code verwenden

Tool aktivieren:

```
#runSubagent
```

Beispiel-Prompt:

```
Analysiere die #file:api mit #runSubagent und empfehle die
beste Authentifizierungsstrategie für einen Web-Client.
```

Was passiert:

1. LLM erstellt einen spezifischen Prompt für den SubAgent
2. SubAgent startet mit **nur** den übergebenen Informationen
3. SubAgent arbeitet unabhängig (Recherche, Analyse, etc.)
4. **Nur das Endergebnis** wird an den Haupt-Chat zurückgegeben
5. Haupt-Chat-Kontext bleibt schlank

Anwendungsfälle für SubAgents

Szenario	Warum SubAgent?
Tiefe Recherche	Sammelt viel Kontext, der den Haupt-Chat überladen würde
Seitenthemen	Kurze Abstecher ohne Haupt-Kontext zu verschmutzen
Technische Analyse	Detailarbeit, die nur ein Fazit liefern soll
Vergleiche	Mehrere Optionen evaluieren, nur Empfehlung zurückgeben
Code-Review	Umfangreiche Analyse, kompaktes Feedback

Beispiele

Recherche zu einer Technologie:

```
Recherchiere mit #runSubagent die Vor- und Nachteile von
Riverpod vs. BLoC für State Management in Flutter.
Gib eine Empfehlung basierend auf unserem Projekt.
```

Architektur-Analyse:

```
Analysiere mit #runSubagent die Abhängigkeiten in #file:src
und identifiziere zirkuläre Imports.
```

Security-Check:

```
Führe mit #runSubagent einen Security-Review der
#file:auth_controller.dart durch.
```

SubAgents vs. Handoffs

Eigenschaft	SubAgents	Handoffs
Kontext	Isoliert (eigener Kontext)	Geteilt (gleicher Chat)
Kontrolle	Automatisch, keine Pause	User kann vor Absenden editieren
Ergebnis	Nur Fazit zurück	Vollständiger Chat-Verlauf
Anwendung	Tiefe Recherche, Seitenthemen	Workflow-Übergabe zwischen Rollen

---

Best Practice: SubAgents effektiv einsetzen

Do	Don't
SubAgents für isolierte Teilaufgaben nutzen	Alles im Haupt-Chat erledigen
Spezifische, fokussierte Prompts formulieren	Vage Anweisungen geben
Kontext explizit übergeben (#file:...)	Annehmen, dass SubAgent alles weiß
Bei langen Chats regelmäßig SubAgents nutzen	Kontext unbegrenzt wachsen lassen

> 💡 **Tipp:** Nutze SubAgents als "Consultants" – sie recherchieren im Hintergrund und liefern nur das Ergebnis.

Quellen

Quelle	Link
VS Code SubAgents Docs	code.visualstudio.com/.../chat-sessions#subagents
Unified Agent Experience Blog	code.visualstudio.com/blogs/.../unified-agent-experience
Context Engineering	dbreunig.com/.../how-to-fix-your-context

1.6.4. Prompt-Bibliotheken

Was sind Prompt-Bibliotheken?

Prompt-Bibliotheken sind kuratierte Sammlungen wiederverwendbarer Prompt-Vorlagen (Textbausteine) für wiederkehrende Aufgaben in der Softwareentwicklung. Sie standardisieren die Zusammenarbeit mit GitHub Copilot Chat, erhöhen die Qualität, beschleunigen wiederkehrende Schritte (z. B. Tests generieren, refaktorieren, dokumentieren) und erleichtern Onboarding und Konsistenz im Team.

Wesentliche Bausteine einer Prompt-Vorlage:

- Ziel: Klare Aufgabe/Outcome (z. B. „Unit-Tests für Funktion X generieren“)
- Kontext: Welche Artefakte/Linien sind relevant (z. B. `#file`, `#selection`, `@workspace`)
- Anforderungen: Stil, Constraints, Akzeptanzkriterien
- Beispiele: Ein- und Ausgabe-Beispiele, wenn sinnvoll
- Grenzen/Sicherheit: Was nicht tun (z. B. keine Secrets, keine externen APIs ohne Freigabe)

Hinweis: GitHub stellt mit dem „Copilot Chat Cookbook“ offizielle, thematisch sortierte Beispiel-Prompts bereit, und die Seite „Prompt engineering for GitHub Copilot Chat“ liefert Strategien, wie Prompts strukturiert und präzisiert werden können. Siehe Quellen unten.

Nutzen in der Softwareentwicklung

Vorteil	Beschreibung
Konsistenz	Gleiche Aufgaben werden mit gleichbleibender Qualität erledigt

Vorteil	Beschreibung
Effizienz	Weniger Tipparbeit und schnelleres Arbeiten mit Copilot Chat
Qualität	Explizite Kriterien (Tests, Stil, Constraints) verbessern Ergebnisse
Wissensweitergabe	Best Practices werden über Vorlagen teamweit geteilt

---

Verzeichnisstruktur im Repository

Empfehlung: Prompts versionieren und nahe am Code ablegen.

```
.github/
└── prompts/
    ├── communicate/              # Kommunikations- & Template-Prompts
    ├── testing/                  # Tests generieren, Testdaten, Coverage
    ├── refactoring/              # Lesbarkeit, Maintainability, Patterns
    ├── debugging/                # Fehlersuche, Log-Analyse
    └── docs/                     # README, ADR, Changelogs
```

Jede Datei enthält eine einzelne Prompt-Vorlage. Metadaten helfen bei Auffindbarkeit und Governance.

Vorlage: Prompt-Datei mit Metadaten

```
---
title: Generate Unit Tests (Copilot Chat)
category: testing
intent: create-tests-for-selected-function
context: "#selection, #file"
environment: "VS Code / github.com Copilot Chat"
owner: team-backend
version: 1.0
---

Ziel:
- Erzeuge aussagekräftige Unit-Tests für die markierte Funktion in #selection.

Rahmenbedingungen:
- Nutze das bestehende Test-Framework des Projekts.
- Decke Randfälle ab; benenne Tests sprechend.
- Erzeuge nur Code; keine Erklärtexte.

Artefakte/Kontext:
- Relevante Datei: #file
- Markierter Ausschnitt: #selection

Akzeptanzkriterien:
```

```
- Tests sind deterministisch, unabhängig und leicht wartbar.
- Kein Zugriff auf Netzwerk/Dateisystem außer Mocks/Stubs.
```

Die Verwendung von Chat-Variablen (`#file`, `#selection`, `#workspace`) und Teilnehmern (z. B. `@workspace`) ist in den GitHub Docs zu Copilot Chat beschrieben. So wird der Prompt präzise an den aktiven Code-Kontext gebunden.

Beispiele für Team-Prompts (Auszug)

1. Tests generieren (angelehnt an das Copilot Chat Cookbook):

```
Erzeuge Unit-Tests für die markierte Funktion in #selection.
Anforderungen:
- Nutze [Framework des Projekts] und übliche Arrange-Act-Assert-Struktur.
- Erzeuge Grenzfall- und Fehlerfall-Tests.
- Keine externen I/O-Operationen.
Kontext: #file, #selection
Output: Nur Testcode.
```

2. Refactoring-Vorschläge (Lesbarkeit & Maintainability):

```
Analysiere die Funktion(en) in #selection auf Lesbarkeit, Komplexität und
Maintainability.
Liefere eine kurze, geordnete Liste konkreter Refactoring-Schritte
(z. B. Extrahieren, Umbenennen, Guard Clauses, Pattern-Einsatz).
Kontext: #file, #selection
Output: Kurze Begründungen + Code-Snippets für die wichtigsten Schritte.
```

3. README-Abschnitt generieren:

```
Erzeuge einen README-Abschnitt "Usage" basierend auf den öffentlichen APIs in
#file.
Anforderungen: Kurzer Überblick, Codebeispiele, Hinweise zu Grenzen und Version.
Output: Markdown-Abschnitt ohne Einleitung/Outro.
```

Qualitätssicherung und Governance

- Metadaten: `owner`, `version`, `category`, `intent`, `last-reviewed`
- Review-Prozess: Pull-Requests wie bei Code; kurze Testläufe im Chat
- De-Duplizierung: Ähnliche Prompts zusammenführen; veraltete Vorlagen deprecaten
- Sicherheit: Keine Secrets im Prompt; keine Aufforderung, Policies zu umgehen
- Kontextklarheit: Präzise Nutzung von `#selection`/`#file` statt vager Verweise

---

Integration in Tools/Workflows

- VS Code: Prompts als Markdown-Dateien im Repo pflegen und bei Bedarf in Copilot Chat einfügen; mit `#file`/`#selection` anreichern.
- GitHub Docs „Getting started with prompts for Copilot Chat“ zeigt Schlüsselwörter (z. B. `@workspace`, Slash-Commands, Chat-Variablen) zur gezielten Kontextübergabe.
- Externe Bibliotheken: PromptHub bietet eine kuratierte Sammlung und „Open in Copilot“ für github.com, womit Prompts direkt in Copilot Chat vorbefüllt werden können (siehe Quelle unten). Eignet sich als Inspiration; teaminterne Bibliothek bleibt das „Source of Truth“.

Do / Don’t (kompakt)

Do	Don’t
„Breit starten, dann konkret werden“ (Ziel, Anforderungen, Beispiele)	Vage, kontextlose Einzeiler
Beispiele und Akzeptanzkriterien hinzufügen	Ergebnisse offen lassen
Chat-Variablen (`#file`, `#selection`) nutzen	Auf impliziten Kontext hoffen
Offizielle Docs/Pattern referenzieren	Unsichere Fremdquellen priorisieren
Versionieren und reviewen	Einmal schreiben, nie aktualisieren

---

Quellen (kuratierte Auswahl)

- GitHub Docs – Copilot Chat Cookbook (Beispiel-Prompts): https://docs.github.com/en/copilot/example-prompts-for-github-copilot-chat
- GitHub Docs – Prompt engineering for GitHub Copilot Chat: https://docs.github.com/en/copilot/concepts/prompting/prompt-engineering
- GitHub Docs – Getting started with prompts for Copilot Chat: https://docs.github.com/copilot/get-started/getting-started-with-prompts-for-copilot-chat
- GitHub Docs – Best practices for using GitHub Copilot: https://docs.github.com/en/copilot/get-started/best-practices
- Microsoft Learn – Introduction to prompt engineering with GitHub Copilot: https://learn.microsoft.com/en-us/training/modules/introduction-prompt-engineering-with-github-copilot/
- PromptHub – One-Click GitHub Copilot Prompts (Open in Copilot): https://www.prompthub.us/blog/prompthub-github-one-click-github-copilot-prompts-are-live

---

1.6.5. Komposition von Agentic Workflows

Der größte Nutzen entsteht durch die **Kombination** dieser Werkzeuge zu anspruchsvollen "Agentic Workflows". Ein bewährter Ansatz ist die Trennung von **Planung** und **Implementierung**:

Planung und Implementierung

1. **Planung (Premium-Modell & Prompt File):** Nutzung einer `/plan`-Prompt-Datei mit einem leistungsstarken Modell (z. B. Claude Opus), um einen detaillierten Plan zu erstellen. Dieser Plan wird in einer separaten Datei gespeichert (z. B. `implementation-plan.md`), die alle Schritte und Code-Schnipsel enthält.
2. **Implementierung (Implement Agent & Günstiges Modell):** Start einer neuen Chat-Sitzung mit einem **Implement Agent**, der auf einem effizienteren Modell läuft. Dieser Agent nutzt den erstellten Plan als Kontext und führt die Schritte systematisch aus.

Dieser Workflow optimiert sowohl die Kosten als auch die Qualität der Ergebnisse, da die "Denk"-Arbeit vom teuren Modell übernommen wird, während die Ausführung effizient erfolgt.

1.6.6. Weitere Best Practices

Dieser Abschnitt kann um weitere funktionale Erweiterungen ergänzt werden, z.B.:

- **Prompt-Bibliotheken** – Wiederverwendbare Prompt-Bausteine
- **Agent-Orchestrierung** – Komplexe Multi-Agent-Szenarien
- **Fehlerbehandlung** – Strategien für robuste Agent-Antworten
- **Caching-Strategien** – Häufig benötigte Informationen vorhalten