GitHub Spec Kit - Cheat Sheet

Von chaotischen Prompts zu strukturiertem Code

🎯 Was ist GitHub Spec Kit?

GitHub Spec Kit verwandelt chaotische KI-Prompts in einen strukturierten, überprüfbaren Entwicklungsworkflow und löst das Problem des "Vibe Coding".

Das Problem: Vibe Coding

- KI-generierter Code sieht perfekt aus, versagt aber in der Praxis
- Sprachmodelle müssen Tausende von Details erraten
- Mangel an Spezifikationsklarheit führt zu funktionslosem Code

Die Lösung: Spezifikationsgetriebene Entwicklung

- **Erst spezifizieren, dann codieren** (nicht umgekehrt)
- Präzise Spezifikation als "einzige Quelle der Wahrheit"
- Eliminierung von Rätselraten bei KI-Tools

📋 Voraussetzungen & Installation

Systemanforderungen

- Linux/macOS (oder WSL2 auf Windows)
- Python 3.11+
- Git
- uv (für Package Management)

KI-Framework (eines auswählen)

- **GitHub Copilot**
- **Claude Code**
- **Gemini CLI**

Installation

```
# Projekt initialisieren
uvx --from git+https://github.com/github/spec-kit.git specify init
<PROJECT_NAME>

# Im aktuellen Verzeichnis
specify init --here
```

```
# Mit spezifischem KI-Agent
specify init <PROJECT_NAME> --ai claude
specify init <PROJECT_NAME> --ai gemini
specify init <PROJECT_NAME> --ai copilot

# Tools-Check ignorieren (optional)
specify init <PROJECT_NAME> --ai claude --ignore-agent-tools
```

🏗️ Die 4 Entwicklungsphasen

Phase 1: SPECIFY - Was soll gebaut werden?

Ziel: User Journeys und gewünschte Ergebnisse definieren

Befehl: `/specify`

Fokus:

- Allgemeine Projektbeschreibung
- Grundlegende Funktionen
- Einfache User Journey
- NICHT den Tech-Stack (das kommt später!)

Beispiel-Prompt:

```
Build an application that can help me organize my photos in separate photo albums.
Albums are grouped by date and can be re-organized by dragging and dropping on the main page.
Albums never contain other nested albums. Within each album, photos are previewed in a tile-like interface.
```

Ergebnis:

- Neue Entwicklungs-Branch wird erstellt
- `spec.md` Datei mit User Stories
- Akzeptanzszenarien und Edge Cases
- Funktionale Anforderungen

Phase 2: PLAN - Wie soll es gebaut werden?

Ziel: Tech-Stack und architektonische Einschränkungen definieren

Befehl: `/plan`

Fokus:

- Gewünschte Frameworks
- Technische Constraints
- Implementierungsdetails

Beispiel-Prompt:

```
The application uses Vite with minimal number of libraries.
Use vanilla HTML, CSS, and JavaScript as much as possible.
Images are not uploaded anywhere and metadata is stored in a local SQLite
database.
```

Ergebnis:

- Detaillierter technischer Plan
- Datenmodell
- Recherche-Dokument mit Framework-Begründungen
- Alternative Lösungsansätze

Phase 3: TASKS - Aufgaben definieren

Ziel: Spezifikation und Plan in ausführbare Aufgaben zerlegen

Befehl: `/tasks`

Fokus:

- Überschaubare, testbare Einheiten
- Eindeutige Nummerierung
- Schrittweise Ausführung

Beispiel:

```
Create MVP version with basic photo organization features
```

Ergebnis:

- Detaillierte Aufgabenliste
- Eindeutige Aufgabennummern (t001, t002, etc.)
- Klare Schritt-für-Schritt Anweisungen

Phase 4: IMPLEMENT - Code schreiben

Ziel: Strukturierte Implementierung basierend auf Spezifikationen

Befehl: `implement`

Verwendung:

```
implement t001           # Spezifische Aufgabe
implement t001 t002 t003 # Mehrere Aufgaben
implement phase 1        # Ganze Phase
```

Vorteile:

- Iterative Entwicklung
- Fortschrittsverfolgung

- Überprüfung vor Implementierung
- Sauberer, strukturierter Code

🚀 Praktischer Workflow

1. Projekt Setup

```
# Terminal öffnen
specify init mein-projekt --ai claude
cd mein-projekt

# VS Code öffnen
code .

# KI-Agent starten (z.B. Claude)
claude
```

2. Spezifikation erstellen

```
/specify [Ihre Projektbeschreibung hier]
```

3. Technischen Plan entwickeln

```
/plan [Tech-Stack und architektonische Details]
```

4. Aufgaben generieren

```
/tasks [Beschreibung der gewünschten Features]
```

5. Implementation

```
implement t001 t002 t003
```

💡 Vor- und Nachteile

✅ Vorteile von Spec Kit

- **Strukturierter Ansatz:** Eliminiert chaotisches "Vibe Coding"
- **Klarheit:** Eindeutige Spezifikationen vermeiden Missverständnisse
- **Kontrolle:** Schrittweise Überprüfung und Iteration möglich
- **Qualität:** Sauberer, zuverlässiger Code
- **Flexibilität:** Funktioniert mit verschiedenen KI-Tools
- **Nachverfolgbarkeit:** Klare Dokumentation aller Entwicklungsschritte

⚠️ Nachteile/Herausforderungen

- **Lernkurve:** Neuer Ansatz erfordert Umdenken
- **Zeitaufwand:** Initiale Spezifikation dauert länger

- **Tool-Abhängigkeit:** Benötigt spezifische KI-Frameworks
- **Komplexität:** Mehr Schritte als direktes Coding
- **Experimentell:** Noch in der Entwicklungsphase

🔧 Troubleshooting

Häufige Probleme & Lösungen

Git-Authentifizierung auf Linux:

```
# Git Credential Manager installieren
wget https://github.com/git-ecosystem/git-credential-
manager/releases/download/v2.6.1/gcm-linux_amd64.2.6.1.deb
sudo dpkg -i gcm-linux_amd64.2.6.1.deb
git config --global credential.helper manager
```

KI-Agent nicht erkannt:

```
# Mit --ignore-agent-tools flag umgehen
specify init <PROJECT_NAME> --ai claude --ignore-agent-tools
```

Spezifikation zu ungenau:

- Mehr Details zur User Journey hinzufügen
- Edge Cases explizit erwähnen
- Funktionale Anforderungen präzisieren

Plan zu komplex:

- KI nach Vereinfachung fragen
- Über-Engineering identifizieren und entfernen
- Constitution-Dokument beachten

Implementation-Fehler:

- Build-Logs in KI-Tool kopieren
- Browser-Fehler manuell übertragen
- Schrittweise Fehlersuche

🌟 Best Practices

Für die SPECIFY Phase:

- **Seien Sie explizit:** Je detaillierter, desto besser
- **Fokus auf "Was" und "Warum":** Nicht auf "Wie"
- **User Stories schreiben:** Aus Nutzersicht denken
- **Edge Cases erwähnen:** Randbereich-Szenarien berücksichtigen

Für die PLAN Phase:

- **Tech-Stack begründen:** Warum diese Frameworks?
- **Constraints definieren:** Technische Einschränkungen klar kommunizieren
- **Recherche nutzen:** KI-generierte Begründungen prüfen
- **Alternativen bewerten:** Verschiedene Ansätze dokumentieren

Für die TASKS Phase:

- **Kleine Schritte:** Überschaubare Aufgaben definieren
- **Testbarkeit:** Jede Aufgabe sollte überprüfbar sein
- **Reihenfolge beachten:** Logische Abhängigkeiten berücksichtigen
- **MVP-Ansatz:** Mit Minimum Viable Product starten

Für die IMPLEMENT Phase:

- **Schrittweise vorgehen:** Nicht alle Aufgaben auf einmal
- **Regelmäßig testen:** Nach jeder Aufgabe Funktionalität prüfen
- **Dokumentation aktuell halten:** Änderungen in Spezifikation reflektieren
- **Code Review:** Generierte Implementierung kritisch prüfen

---

📚 Weiterführende Ressourcen

Offizielle Dokumentation

- **GitHub Repository:** https://github.com/github/spec-kit/
- **Issues & Support:** https://github.com/github/spec-kit/issues/new

Development Tools

- **uv Package Manager:** https://docs.astral.sh/uv/
- **Python Downloads:** https://www.python.org/downloads/
- **Git:** https://git-scm.com/downloads

⚡ Quick Reference - Wichtigste Befehle

```
# Projekt Setup
specify init <PROJECT_NAME> --ai <AGENT>

# Die 4 Hauptbefehle
/specify    # Was soll gebaut werden?
/plan       # Wie soll es gebaut werden?
/tasks      # Aufgaben definieren
implement   # Code schreiben

# Implementation Beispiele
implement t001
```

```
implement t001 t002 t003
implement phase 1
```

🎓 Fazit

GitHub Spec Kit revolutioniert die KI-gestützte Entwicklung durch:

- **Strukturierten Ansatz** statt chaotischem Vibe Coding
- **Klare Spezifikationen** als Grundlage für zuverlässigen Code
- **Iterative Entwicklung** mit vollständiger Kontrolle
- **Bessere Code-Qualität** durch durchdachte Planung

Nächste Schritte:

1. Voraussetzungen installieren
2. Erstes Projekt mit `/specify` starten
3. Schrittweise durch die 4 Phasen arbeiten
4. Best Practices anwenden und iterieren