

Mein neues Werkzeug: Claude Code

Aufgabe stellen, loslassen, Ergebnis prüfen

Carsten Grohmann

Unix-Stammtisch Dresden

7. Januar 2026

Agenda

Einführung und Grundlagen
Arbeiten mit Claude Code
Best Practices und Sicherheit
Live-Demos
Praxiserfahrungen
Fazit und Ausblick
Abschluss

Agenda

Agenda

1. Einführung und Grundlagen
2. Arbeiten mit Claude Code
3. Best Practices und Sicherheit
4. Live-Demos
5. Praxiserfahrungen
6. Fazit und Ausblick
7. Abschluss

Einführung und Grundlagen

Warum Claude Code?

Was ist Claude Code?

Das Grundprinzip: Der autonome Entwicklungszyklus
Nutzungslimits Claude Code Pro (20\$/Monat)

Warum Claude Code?

- ▶ Autonome Problemlösung
- ▶ Kontextbewusstes Arbeiten
- ▶ Praktische Integration
- ▶ Transparenz und Effizienz
- ▶ Lernunterstützung

Mein neues Werkzeug: Claude Code

└ Einführung und Grundlagen

└ Warum Claude Code?

└ Warum Claude Code?

Warum Claude Code?

- ▶ Autonome Problemlösung
- ▶ Kontextbewusstes Arbeiten
- ▶ Praktische Integration
- ▶ Transparenz und Effizienz
- ▶ Lernunterstützung

- Autonome Problemlösung:
 - Komplexe Aufgaben selbstständig in Schritte zerlegen
 - Code schreiben, testen, debuggen und iterativ verbessern
- Kontextbewusstes Arbeiten
 - Codebase durchsuchen und verstehen, Architekturmuster erkennen
 - Lösungen passend zum bestehenden Code
- Praktische Integration
 - Git-Integration, Build-Systeme, Tests und Entwicklungstools
 - Web-Recherche und Dokumentation
- Transparenz und Effizienz
 - Jeden Schritt nachvollziehbar
 - Automatisiert repetitive Aufgaben
- Lernunterstützung
 - Zeigt Best Practices und alternative Ansätze
 - Hilft beim Verstehen unbekannter Codebases

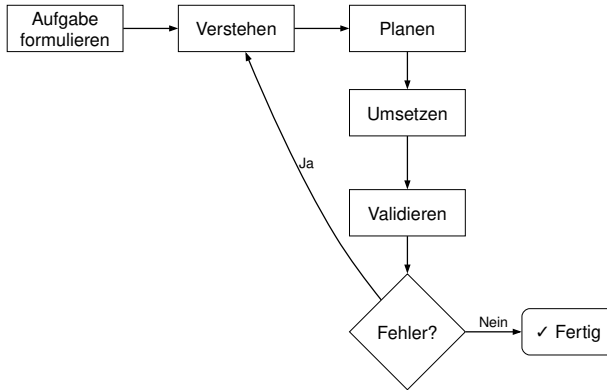
Was ist Claude Code?

- ▶ Offizielles CLI-Tool von Anthropic für Claude AI
- ▶ Terminalbasiertes Entwicklungswerkzeug
- ▶ Direkter Zugriff auf Codebase, Dateisystem und Build-Tools
- ▶ Läuft mit Nutzerrechten (keine Isolierung)
- ▶ Verfügbar für Linux, macOS und Windows

Paradigmenwechsel:

- ▶ Nicht: “Generiere Code für X”
- ▶ Sondern: “Analysiere Codebase, plane Änderung, implementiere, teste”

Das Grundprinzip: Der autonome Entwicklungszyklus



Wichtig: Tests und hohe Testabdeckung sind essenziell.

Mein neues Werkzeug: Claude Code

└ Einführung und Grundlagen

└ Das Grundprinzip: Der autonome Entwicklungszyklus

└ Das Grundprinzip: Der autonome Entwicklungszyklus

Das Grundprinzip: Der autonome Entwicklungszyklus



Wichtig: Tests und hohe Testabdeckung sind essenziell.

- Eigenständige Arbeit nach Aufgabenstellung
- Autonome Umsetzung bis zum getesteten Ergebnis
- Tests und Testabdeckung sind kritisch für Erfolg

Nutzungslimits Claude Code Pro (20\$/Monat)

Sitzungslimit:

- ▶ ~10-40 Prompts bei Claude Code (45 Nachrichten bei claude.ai)
- ▶ Gilt gemeinsam über alle Plattformen (Web, Desktop, Code)
- ▶ Ungenutzte Token verfallen nicht
- ▶ Reset 5 Stunden nach Session-Start

Wochenlimit:

- ▶ Offiziell: 40-80 Stunden Sonnet 4 pro Woche
- ▶ Praxis: 10-20 Stunden (ca. 2-3h pro Arbeitstag)
- ▶ Ungenutzte Token verfallen
- ▶ Stark variabel je nach Codebase-Größe und Auto-Accept Mode

Wichtig:

- ▶ Fehlende Transparenz bei den Messungen
- ▶ Änderungen ohne Ankündigung

Arbeiten mit Claude Code

Die drei Modi

Drei Betriebsmodi für unterschiedliche Arbeitsweisen

- ▶ Ask-Modus - Gezieltes Nachfragen und Klären
- ▶ Plan-Modus - Strukturierte Planung vor der Implementierung
- ▶ Automatic-Modus - Autonome Umsetzung

Ask-Modus: Gezieltes Nachfragen und Klären

Wofür:

- ▶ Verständnisfragen zur Codebase
- ▶ Erklärungen zu bestehendem Code
- ▶ Schnelle Analysen ohne Änderungen

Beispiele:

```
> Wo werden Fehler vom Client behandelt?  
→ Claude analysiert Code und zeigt: "In src/services/process.ts:712"  
  
> Welche regulären Ausdrücke werden in diesem Projekt verwendet?  
→ Claude listet alle Regex-Patterns mit Fundstellen auf
```

Typische Anwendungsfälle:

- ▶ Onboarding in fremde Codebases
- ▶ Code-Reviews verstehen
- ▶ Architekturentscheidungen nachvollziehen
- ▶ “Wie funktioniert Feature X?”

Mein neues Werkzeug: Claude Code

└─ Arbeiten mit Claude Code

└─ Die drei Modi

Typische Anwendungsfälle:

- ▶ Onboarding in fremde Codebases
- ▶ Code-Reviews verstehen
- ▶ Architekturentscheidungen nachvollziehen
- ▶ "Wie funktioniert Feature X?"

- Schnelle Codebase-Analyse in Sekunden
- Präzise Antworten ohne Änderungen
- Ideal für Onboarding und Verständnisfragen

Plan-Modus: Strukturierte Planung vor Implementierung

Wofür:

- ▶ Komplexe Änderungen mit mehreren Schritten
- ▶ Unsicherheit über beste Herangehensweise
- ▶ Klärung von Anforderungen vor Umsetzung

Workflow:

1. Aufgabe beschreiben
2. Claude analysiert Codebase
3. Claude erstellt detaillierten Plan mit:
 - ▶ Betroffene Dateien
 - ▶ Einzelne Schritte
 - ▶ Mögliche Risiken
 - ▶ Alternative Ansätze
4. Plan prüfen und freigeben
5. Umsetzung starten

Praxisbeispiel:

Plan: unittest → Pytest Migration

Claude analysiert:

- ▶ test.py (1,6k Zeilen mit unittest)
- ▶ Makefile (Test-Targets)
- ▶ Dependencies in requirements.txt

Claude schlägt vor:

1. Test-Klassen zu pytest-Funktionen konvertieren
2. setUp/tearDown durch Fixtures ersetzen
3. self.assert* durch assert ersetzen
4. Makefile anpassen
5. Dependencies aktualisieren
6. Tests ausführen und validieren

→ Freigabe → Autonome Implementierung

Mein neues Werkzeug: Claude Code

└─ Arbeiten mit Claude Code

└─ Die drei Modi

Claude schlägt vor:

1. Test-Klassen zu pytest-Funktionen konvertieren
2. setUp/tearDown durch fixtures ersetzen
3. self.assert* durch assert ersetzen
4. Mollie anpassen
5. Dependencies aktualisieren
6. Tests ausführen und validieren

→ Freigabe → Autonome Implementierung

- Wertvoll bei komplexen Refactorings
- Zeigt alle Implikationen vor Code-Änderungen
- Verhindert fehlerhafte Implementierungen durch Vorausplanung

Automatic-Modus: Autonome Umsetzung

Wofür:

- ▶ Klare, gut definierte Aufgaben
- ▶ Änderungen mit vorhandenem Test-Suite
- ▶ Routine-Refactorings

Merkmale:

- ▶ Claude arbeitet selbstständig durch Todo-Liste
- ▶ Führt Tests automatisch aus
- ▶ Korrigiert Fehler eigenständig
- ▶ Stoppt bei unklaren Situationen und fragt nach

Beispiel-Ablauf:

Prompt: Check regular expressions and suggest improvements

Mustertext: "[473206] 504 473206 7572670"

Hinweis: Der RE soll die PID aus dem ersten Feld extrahieren

Claude Code:

1. Scannt REs, identifiziert: `^\[(?P<pid>[\d]+)\]`
2. Ändert zu `^\[(?P<pid>\s*\d+)\]` und testet
3. Tests fehlgeschlagen
4. Analysiert Fehler, korrigiert zu `^\[\s*(?P<pid>\d+)\]`
5. Erneute Tests erfolgreich, abgeschlossen

Mein neues Werkzeug: Claude Code

└─ Arbeiten mit Claude Code

└─ Die drei Modi

Beispiel-Ablauf:

Prompt: Check regular expressions and suggest improvements

Mustertext: "[473206] 504 473206 7572670"

Hinweis: Der RE soll die PID aus dem ersten Feld extrahieren

Claude Code:

1. Scant REs, identifiziert: "\\(?(?pid>[\\d]+)\\)"
2. Ändert zu "\\(?(?pid>\\w+\\d+)\\)" und testet
3. Tests fehlgeschlagen
4. Analysiert Fehler, korrigiert zu "\\(\\w+(?pid>\\d+)\\)"
5. Erneute Tests erfolgreich, abgeschlossen

- Beispiel für eine autonome Änderung aus einer größeren Aufgabe
- inkl. erster fehlerhafter Änderung mit anschließenden Test und Korrektur

Grenzen der Autonomie:

- ▶ Bei Unsicherheiten: Claude fragt nach (Multiple Choice Dialog)
- ▶ Bei fehlenden Informationen: Claude stoppt und bittet um Input
- ▶ Bei kritischen Operationen: Claude warnt vorher

Mein neues Werkzeug: Claude Code

└─ Arbeiten mit Claude Code

└─ Die drei Modi

Grenzen der Autonomie:

- ▶ Bei Unsicherheiten: Claude fragt nach (Multiple Choice Dialog)
- ▶ Bei fehlenden Informationen: Claude stoppt und bittet um Input
- ▶ Bei kritischen Operationen: Claude warnt vorher

- Kernelement: Aufgabe stellen, loslassen, Ergebnis prüfen
- Autonomie mit Kontrolle: Claude stoppt bei Unsicherheiten
- Automatische Fehlerkorrektur und Test-Validierung

Was passiert im Hintergrund:

- Read: Dateien lesen und analysieren
- Grep/Glob: Code durchsuchen
- Edit: Präzise Änderungen vornehmen
- Bash: Tests, Build-Prozesse ausführen
- TodoWrite: Fortschritt tracken

Wann welcher Modus? - Entscheidungshilfe

Situation	Modus	Grund
Analyse/Erklärung	Ask	Keine Änderung
Großes Refactoring	Plan → Auto	Viele Dateien
Einfache Änderung	Automatic	Klar definiert
Performance	Plan	Mehrere Ansätze
Framework-Migration	Plan → Auto	Groß, strukturiert
Bugfix (mit Tests)	Automatic	Klein, testbar
Feature-Evaluierung	Plan	Diskussionsbedarf

Mein neues Werkzeug: Claude Code

└─ Arbeiten mit Claude Code

└─ Die drei Modi

└─ Wann welcher Modus? - Entscheidungshilfe

Wann welcher Modus? - Entscheidungshilfe

Situation	Modus	Grund
Analyse/Erklärung	Ask	Keine Änderung
Großes Refactoring	Plan → Auto	Viele Dateien
Einfache Änderung	Automatic	Klar definiert
Performance	Plan	Mehrere Ansätze
Framework-Migration	Plan → Auto	Groß, strukturiert
Bugfix (mit Tests)	Automatic	Klein, testbar
Feature-Evaluierung	Plan	Diskussionsbedarf

- Modi sind kombinierbar
- Typischer Workflow: Ask → Plan → Automatic
- Modiwechsel je nach Aufgabenphase

Konfigurationsebenen und -verzeichnisse

- ▶ `~/.claude/`: Globale Konfiguration
 - ▶ Gilt für alle Claude Code Sitzungen
 - ▶ Persönliche Präferenzen, Code-Stil, Review-Standards
 - ▶ Beispiel: `~/.claude/CLAUDE.md` mit Coding-Richtlinien
- ▶ `./.claude/`: Projektspezifische Konfiguration
 - ▶ Nur für aktuelles Projektverzeichnis
 - ▶ Überschreibt globale Einstellungen
 - ▶ Projektkonventionen, Build-Anweisungen
 - ▶ Beispiel: `./.claude/CLAUDE.md` mit Architekturentscheidungen

Beispiele

Global (~/.claude/CLAUDE.md)

```
# Global Developer Settings

## Generic Rules
- Comments describe "why", not "how"
- Display line numbers when quoting code

## Code Style
- English for all code, comments, documentation
- Clean, readable code with meaningful names
- Follow DRY principle

## Git
- ASCII only, 50 char subject, 72 char body lines
- Explain what and why (not how)
```

Projektspezifisch - `./.claude/CLAUDE.md` für OOMAnalyser

```
# OOMAnalyser - Project Guide for Claude Code

## Project Overview
Web-based tool analyzing Linux kernel OOM messages

## Technology Stack
- Python 3.7+ / Transcript 3.7 / Rollup

## Transcript Compatibility
- Code runs in Python AND JavaScript
- Avoid: exec, eval, try/except
- Use DOM mock classes for browser API testing

## Commits
- Run make black and all tests before commit
```

Nutzen:

- ▶ Global: Konsistenter Code-Stil projektübergreifend
- ▶ Projektspezifische: Claude kennt Architektur, Build-Prozess, Besonderheiten
- ▶ Wiederverwendbare Anweisungen reduzieren manuelle Erklärungen

Best Practices und Sicherheit

Effektive Modus-Nutzung

Ask-Modus:

- ▶ Für schnelle Analysen und Verständnisfragen nutzen
- ▶ Keine Code-Änderungen, nur Informationen sammeln
- ▶ Ideal beim Onboarding in fremde Codebases

Plan-Modus:

- ▶ Bei >3 Dateien: Immer planen lassen
- ▶ Unklare Anforderungen: Planen + Rückfragen
- ▶ Kritische Änderungen: Plan prüfen, dann implementieren

Automatic-Modus:

- ▶ Nur bei klaren, testbaren Aufgaben nutzen
- ▶ Test-Suite muss vorhanden sein
- ▶ Claude stoppt automatisch bei Unsicherheiten

Kontext-Management: Weniger ist mehr

Kernproblem:

- ▶ Voller Kontext (200k Token) → reduzierte Qualität
- ▶ Fokussierter Kontext → präzisere Antworten
- ▶ Irrelevanter Kontext verschlechtert Ausgabequalität

Strategien:

- ▶ Kurze Sessions, /clear bei Themenwechsel
- ▶ Verwandte Fragen gebündelt stellen (nicht sequenziell)
- ▶ Große Projekte thematisch aufteilen
- ▶ Nur relevante Dateien einbeziehen

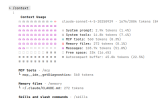
```
> /context
```

Context Usage

MCP tools · /mcp

Memory files - /memory

Skills and slash commands · /skills



- Weniger Kontext → bessere Ergebnisse
- Modell arbeitet präziser mit fokussiertem Kontext
- voller Kontext verschlechtern die Ausgabequalität

Smart Prompting:

- Spezifität: "Optimiere N+1 Queries in user_service.py" statt "Verbessere Performance"
- Prompt-Bausteine: Tests erwähnen, Vollständigkeit fordern, Ursachenanalyse statt Symptombehandlung
- Top-Down bei Komplexität: Beschreiben → Struktur → Review → Ausarbeiten → Final Review
- Fortgeschritten: Referenzpersönlichkeiten nutzen ("Reviewe diese Änderung wie Linus Torvalds es tun würde, mit Fokus auf langfristige Wartbarkeit.")

Prompt-Optimierung: Die KI als Meta-Berater

Prompt-Verbesserung: Claude den Prompt selbst analysieren lassen

Analyze this prompt intended for an AI coding assistant:

"""

[DEIN PROMPT]

"""

Rate 1-5 (with justification):

- Clarity: Is the goal obvious?
- Specificity: Are requirements concrete?
- Completeness: Is context sufficient?
- Grammar: Any language issues?

Provide:

1. Top 3 issues preventing effectiveness
2. Improved rewrite addressing those issues
3. One key insight about what makes it better

Mein neues Werkzeug: Claude Code

└ Best Practices und Sicherheit

└ Prompt-Optimierung: Die KI als Meta-Berater

└ Prompt-Optimierung: Die KI als Meta-Berater

Prompt-Optimierung: Die KI als Meta-Berater

Prompt-Verbesserung: Claude den Prompt selbst analysieren lassen

Analyse dein prompt anhand der an KI making assistant:

```
"""
[DEIN PROMPT]
"""

Rate 0-10 (only justification):
- Clarity: Is the goal clear?
- Specificity: Are requirements concrete?
- Completeness: Is context sufficient?
- Grammar: Any language issues?
```

Provide:

- Top 3 issues preventing effectiveness
- Improve each one addressing those issues
- How you thought about what makes it better

- Prompt-Analyse durch Claude selbst
- Wann: Komplexe Refactorings, unbefriedigende Ergebnisse, vor zeitaufwändigen Operationen
- Nutzen: Verhindert Missverständnisse, reduziert Token-Verbrauch, identifiziert Prompt-Schwächen

Limit-Management: Budget clever nutzen

Sitzungslimit (5-Stunden-Fenster):

- ▶ Komplexe Aufgaben zuerst (volles Budget)
- ▶ Einfache Fixes später
- ▶ Bei Warnung: Kritisches abschließen, dann pausieren

Wochenlimit (praktisch 10-20h):

- ▶ Stapelverarbeitung: Ähnliche Aufgaben bündeln
- ▶ Parallele Tool-Calls nutzen
- ▶ Große Refactorings sorgfältig planen (vermeidet Neuanläufe)

Mein neues Werkzeug: Claude Code

└ Best Practices und Sicherheit

└ Limit-Management: Budget clever nutzen

└ Limit-Management: Budget clever nutzen

Limit-Management: Budget clever nutzen

Sitzungslimit (5-Stunden-Fenster):

- ▶ Komplexe Aufgaben zuerst (volles Budget)
- ▶ Einfache Fixes später
- ▶ Bei Warnung: Kritisches abschließen, dann pausieren

Wochenlimit (praktisch 10-20h):

- ▶ Stapelverarbeitung: Ähnliche Aufgaben bündeln
- ▶ Parallele Tool-Calls nutzen
- ▶ Große Refactorings sorgfältig planen (vermeidet Neuanläufe)

- Limits: Oft knapper als offizielle Angaben
- Strategische Session-Planung erforderlich
- Neuanläufe verdoppeln Token-Verbrauch

Sicherheit: Umgang mit Geheimnissen

Risiko: Datenübertragung in die Cloud

- ▶ Code und Dateien werden an Anthropic-Server übertragen
- ▶ Daten standardmäßig für Training verwendet (opt-out möglich)
- ▶ Gilt auch für .env-Dateien, Konfigurationen, Commit-Historie

Best Practices:

- ▶ Keine Credentials in Code committen
- ▶ .gitignore konsequent nutzen (.env, credentials.json, etc.)
- ▶ Umgebungsvariablen oder Secret-Manager verwenden
- ▶ Vor Verwendung: Projektverzeichnis auf Geheimnisse prüfen

Mein neues Werkzeug: Claude Code

└ Best Practices und Sicherheit

└ Sicherheit: Umgang mit Geheimnissen

└ Sicherheit: Umgang mit Geheimnissen

Sicherheit: Umgang mit Geheimnissen

Risiko: Datenübertragung in die Cloud

- ▶ Code und Dateien werden an Anthropic-Server übertragen
- ▶ Daten standardmäßig für Training verwendet (opt-out möglich)
- ▶ Gilt auch für `.env`-Dateien, Konfigurationen, Commit-Historie

Best Practices:

- ▶ Keine Credentials in Code committen
- ▶ `gitignore` konsequent nutzen (`.env`, `credentials.json`, etc.)
- ▶ Umgebungsvariablen oder Secret-Manager verwenden
- ▶ Vor Verwendung: Projektverzeichnis auf Geheimnisse prüfen

- Alle gelesenen Dateien werden in die Cloud übertragen
- Claude Pro: Standardmäßig fürs Training verwendet, opt-out möglich
- Claude for Work/Enterprise: Nicht fürs Training verwendet
- Besonders kritisch: API-Keys, Passwörter, Tokens, private Keys

Sicherheit: Sandbox-Betrieb

Sicherheitsrisiko:

- ▶ Claude Code läuft mit vollen Nutzerrechten
- ▶ Kann Shell-Befehle ausführen (`rm -rf`, destruktive Make-Targets)
- ▶ Zugriff auf gesamtes Home-Verzeichnis

Absicherungsmöglichkeiten unter Linux:

Tool	Isolationsebene	Komplexität
Bubblewrap	Projektverzeichnis	Mittel
Firejail	Vordefinierte Profile	Niedrig
Docker	Container	Hoch

Bubblewrap-Beispiel:

```
bwrap \  
--ro-bind /usr /usr --ro-bind /lib /lib \  
--bind "$HOME/.claude" "$HOME/.claude" \  
--bind "$(pwd)" "$(pwd)" \  
--chdir "$(pwd)" --unshare-all --share-net \  
claude
```

Mein neues Werkzeug: Claude Code

- └ Best Practices und Sicherheit
 - └ Sicherheit: Sandbox-Betrieb

Absicherungsmöglichkeiten unter Linux:

Tool	Isolationsebene	Komplexität
Bubblewrap	Projektverzeichnis	Mittel
Firejail	Vordefinierte Profile	Niedrig
Docker	Container	Hoch

Bubblewrap-Beispiel:

```
#!/bin/bash
set -e
set -x

# Create a new project directory
mkdir -p ~/projects/secure

# Create a new project directory
cd ~/projects/secure

# Create a new project directory
mkdir -p ~/projects/secure
```

- Claude Code läuft mit vollen Nutzerrechten
- Git-basiertes Arbeiten: Pflicht
- Sandboxing (Bubblewrap/Firejail): Empfohlen

Lizenz und Urheberrecht

Kernfrage: Wem gehört der von Claude Code generierte Code?

Antwort: Der Code gehört dem Nutzer

Rechtliche Grundlage:

Für Claude Free, Pro, Max: Consumer Terms of Service¹:

“Vorbehaltlich Ihrer Einhaltung unserer Bedingungen übertragen wir Ihnen alle unsere Rechte, Titel und Ansprüche (falls vorhanden) an Outputs.”

Training: Daten werden standardmäßig genutzt, opt-out möglich

¹<https://www.anthropic.com/legal/consumer-terms>

Mein neues Werkzeug: Claude Code

└ Best Practices und Sicherheit

└└ Lizenz und Urheberrecht

└└└ Lizenz und Urheberrecht

Lizenz und Urheberrecht

Kernfrage: Wem gehört der von Claude Code generierte Code?

Antwort: Der Code gehört dem Nutzer

Rechtliche Grundlage:

Für Claude Free, Pro, Max: Consumer Terms of Service¹.

"Vorbehaltlich Ihrer Einhaltung unserer Bedingungen übertragen wir Ihnen alle unsere Rechte, Titel und Ansprüche (falls vorhanden) an Outputs."

Training: Daten werden standardmäßig genutzt, opt-out möglich

¹<https://www.anthropic.com/legal/consumer-terms>

- Consumer Terms gelten für Claude Pro (nicht Commercial Terms für Business-Pläne)
- Englischer Originaltext: "Subject to your compliance with our Terms, we assign to you all of our right, title, and interest—if any—in Outputs."
- Business-Kunden (Claude for Work/Enterprise/API) vom Training ausgenommen
- Code unter Projektlizenz veröffentlicherbar (MIT, GPL, etc.)
- Claude Code = Werkzeug (wie IDE, Compiler)

Agenda
Einführung und Grundlagen
Arbeiten mit Claude Code
Best Practices und Sicherheit
Live-Demos
Praxiserfahrungen
Fazit und Ausblick
Abschluss

Umfang des Testprojekts
Demo 1: Regex-Optimierung (einfach, schnell)
Demo 2: Unittest → Pytest Migration (mittelschwer)
Demo 3: Dictionary-Refactoring mit Fragen
Demo 4: Backtracking Sudoku Solver

Live-Demos

Live-Demos

Demonstration des kompletten Zyklus:

Aufgabe → Codebase-Analyse → Plan erstellen →
Implementierung → Tests ausführen → Fehleranalyse →
Korrektur → Erneute Tests

Umfang des Testprojekts

Demo 1: Regex-Optimierung (einfach, schnell)
Demo 2: Unittest → Pytest Migration (mittelschwer)
Demo 3: Dictionary-Refactoring mit Fragen
Demo 4: Backtracking Sudoku Solver

Umfang des Testprojekts

Zweck	Anzahl der Zeilen
1 Datei Python-Sourcecode	6,3k
1 Datei HTML-Sourcecode	1,8k
1 Datei Python-Unittests	1,6k
1 Datei Makefile	140
1 Datei MIT-Lizenz	40
Sonstiges	269

Demo 1: Regex-Optimierung (einfach, schnell)

- ▶ Themen: Codeanalyse, Fehleridentifikation, iteratives Fixen
- ▶ Dauer: 5-8 min
- ▶ Auto-Accept Mode: aus
- ▶ Vorteil: Klarer Vorher-/Nachher-Vergleich

Prompt

```
Separate optional leading whitespace handling from PID capture group in the REC_PROCESS_LINE regular expression in OOMAnalyser.py to ensure only numeric digits are captured in the pid group.
```


Demo 2: Unittest → Pytest Migration (mittelschwer)

- ▶ Themen: Strukturelles Refactoring, Planung, Durchführung
- ▶ Dauer: 45-60 min
- ▶ Auto-Accept Mode: ein
- ▶ Vorteil: Demonstriert planbasierte autonome Entwicklung
- ▶ Zwischenschritte sichern mit `git commit --amend`

Prompt

Create a detailed migration plan to change unit tests from Python's unittest module to pytest in @test.py.

Requirements:

- Break the migration into phases (setup, assertions, fixtures, etc.)
- Ensure all tests run successfully after each phase
- Document the plan in .claude/plans/pytest.md with:
 - Phase descriptions and tasks
 - Success criteria for each phase
 - Timestamp tracking
- After creating the plan, execute each phase incrementally with my approval
- Ask me questions if you need clarification on scope or approach

Goal: Complete migration with no unittest dependencies remaining, using pytest fixtures and parametrization where appropriate.

Demo 3: Dictionary-Refactoring mit Fragen

- ▶ Themen: Interaktive Stärke, Multiple-Choice-Dialoge
- ▶ Dauer: 10-15 min
- ▶ Auto-Accept Mode: ein

Prompt:

Create a todo list to consolidate browser test configuration variables in test.py.

Current structure (test.py:146-178 in BaseInBrowserTests):

```
```python
check_results_gfp_mask: str = ""
check_results_proc_name: str = ""
check_results_proc_pid: str = ""
... ~15 more individual class variables
```
```

Target structure:

```
```python
check_results: Dict[str, str] = {
 'gfp_mask': '',
 'proc_name': '',
 'proc_pid': '',
 # ... consolidated into single dict
}
```
```

Prompt (Fortsetzung):

Requirements:

- Identify all `check_results_*` class variables in `BaseInBrowserTests`
- Only modify variables with string values
- Design the new dictionary structure with appropriate keys
- Update the `check_all_results()` method (`test.py:188-324`) to use dict lookups instead of direct attribute access
- Update child classes (`TestBrowserArchLinux`, etc.) that override these values
- Ensure all 11 tests still pass after refactoring
- After creating the todo list, execute each item incrementally with my approval

Please review this understanding and ask clarifying questions if I've misinterpreted the goal.

Demo 4: Backtracking Sudoku Solver

- ▶ Themen: Gemeinsames Entwickeln eines Sudoku Solvers in Python
- ▶ Dauer: Variable

Praxiserfahrungen

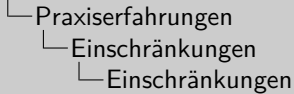
Was funktioniert gut

- ▶ Strukturelles Refactoring mit klarem Ziel
- ▶ Iteratives Bug-Fixing mit Tests
- ▶ Codebase-Analyse und Dokumentation
- ▶ Framework-Migrationen

Einschränkungen

- ▶ Fehlendes Kontextwissen: Was implizit klar ist, muss Claude explizit mitgeteilt werden (z.B. dateiübergreifende Umbenennungen)
- ▶ Unvollständige Testabdeckung wird nicht automatisch erkannt
- ▶ Token-Limits pro Sitzung beachten
- ▶ Maximale Kontextlänge: 200k Token

Mein neues Werkzeug: Claude Code



Einschränkungen

- ▶ Fehlendes Kontextwissen: Was implizit klar ist, muss Claude explizit mitgeteilt werden (z.B. dateiübergreifende Umbenennungen)
- ▶ Unvollständige Textabdeckung wird nicht automatisch erkannt
- ▶ Token-Limits pro Sitzung beachten
- ▶ Maximale Kontextlänge: 200k Token

- “Claude Code ist dumm” - warum explizite Anweisungen erforderlich sind:
 - Umbenennung mehrerer Bezeichner: Jeder Bezeichner muss explizit in allen Dateien umbenannt werden, bevor der nächste folgt - sonst arbeitet Claude dateiweise mit inkonsistenten Zwischenständen
 - Beispiel “Aktualisiere Agenda Zeile 28”: Mensch vergleicht intuitiv Menüpunkte mit Agenda, Claude benötigt diese Schritte explizit
- Token-Limits in der Praxis: Bei langlaufenden Aufgaben mit 145% “überreizt” → Automatische Kontextkomprimierung (Compaction) funktioniert danach nicht mehr

Zeitaufwand im Plan-Modus

- ▶ Planung oft zeitaufwändiger als Implementierung
- ▶ Trade-off: Gründliche Planung vs. schnelles Prototyping
- ▶ Kleine Änderungen: Automatic-Modus effizienter
- ▶ Komplexe Refactorings: Planung zahlt sich aus

Fazit und Ausblick

Fazit und Ausblick

Fazit

- ▶ kann Code über dem eigenen Verständnisniveau schreiben
- ▶ Produktivitätssteigerung
- ▶ Ähnlich Pair-Programming
- ▶ Grenzen kennen, gezielt einsetzen

Ausblick

- ▶ Plugins
- ▶ Subagents
- ▶ MCP-Server: Integration externer Tools (DBs, APIs, IDEs)
- ▶ Erweiterte Automatisierung
 - ▶ Bugreports automatisch analysieren, Duplikate erkennen, Fixes schreiben und testen

Mein neues Werkzeug: Claude Code

└─Fazit und Ausblick

└─Fazit und Ausblick

Fazit und Ausblick

Fazit

- ▶ kann Code über dem eigenen Verständnisniveau schreiben
- ▶ Produktivitätssteigerung
- ▶ Ähnlich Pair-Programming
- ▶ Grenzen kennen, gezielt einsetzen

Ausblick

- ▶ Plugins
- ▶ Subagents
- ▶ MCP-Server: Integration externer Tools (DBs, APIs, IDEs)
- ▶ Erweiterte Automatisierung
 - ▶ Bugreports automatisch analysieren, Duplikate erkennen, Fixes schreiben und testen

- Erweiterte Workflows durch spezialisierte Subagents
- Automatisierung wiederkehrender Entwicklungsaufgaben
- Subagents: Spezialisierte AI-Assistenten für bestimmte Aufgabentypen mit eigenen Kontext (spart Token im Hauptgespräch)

Agenda
Einführung und Grundlagen
Arbeiten mit Claude Code
Best Practices und Sicherheit
Live-Demos
Praxiserfahrungen
Fazit und Ausblick
Abschluss

Fragen und Diskussion
Lizenz

Abschluss

Fragen und Diskussion

- ▶ Fragen
- ▶ Anregungen

Lizenz



Dieses Werk ist lizenziert unter einer “Creative Commons Namensnennung - Nicht-kommerziell - Weitergabe unter gleichen Bedingungen 4.0 International Lizenz”².

²<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.de>