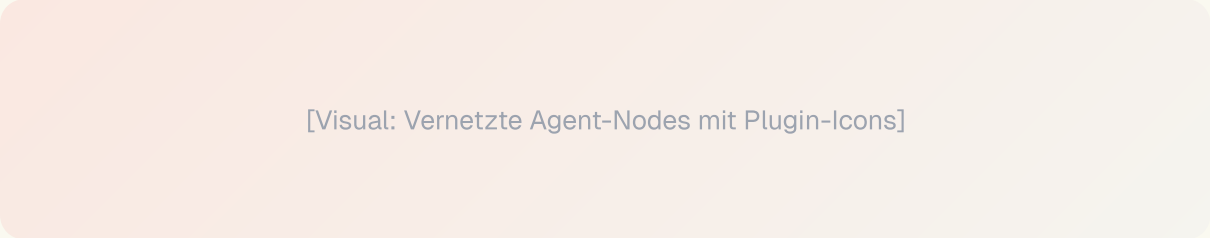
Claude Code Agent Spickzettel: 7 Bewährte Setups für Teams

Mach Claude Code vom Einzeltool zur Team-Superkraft — Plugins, Agents und Architekturen, die wöchentlich Stunden sparen

Von Julian, codecoast labs GmbH — 500+ Stunden Claude Code im Praxistest



Kostenlos für E-Mail-Abonnenten. Skalierbare Setups ohne Chaos.

julian@codecoast.ch

Das Team-Chaos-Problem

Du kennst das: Ein Entwickler entdeckt Claude Code und wird 3x produktiver. Aber wenn du das aufs ganze Team skalieren willst? Chaos.

> **Typische Team-Probleme**
>
> - **Inkonsistente Setups** — Jeder Dev hat eigene Prompts, Workflows und "Tricks"
> - **Kein gemeinsamer Hebel** — Wissen bleibt in Silos; kein Compound-Effekt
> - **Langsames Onboarding** — Neue Mitarbeiter brauchen Wochen für die ungeschriebenen Regeln
> - **Sicherheitsbedenken** — Keine Governance darüber, was die KI darf
> - **Context Switching** — Tools kommunizieren nicht; Claude kennt deinen Stack nicht

Meine Geschichte

Nach 500+ Stunden Praxistests mit Claude Code in Beratungsprojekten habe ich destilliert, was funktioniert — in 7 wiederverwendbare Setups. Das ist keine Theorie — es sind produktionserprobte Architekturen, die Engineering-Teams täglich nutzen.

Was du bekommst

Dieser Spickzettel kodiert Best Practices für Plugins, Agents und Integrationen. Jedes Setup enthält:

- Das spezifische Problem, das es löst
- Architektur-Diagramm
- Copy-Paste Code-Snippets
- ROI-Metriken zur Rechtfertigung der Investition

Team-Audit für €997 buchen →

1 Multi-Agent Code Review System

Problem: Senior Devs versinken in PRs

Deine Senior Engineers verbringen 40% ihrer Zeit mit Code Reviews. Das meiste davon ist das Finden derselben Probleme: Style-Verstöße, fehlende Tests, Sicherheitslücken. Was, wenn KI die repetitiven Checks übernimmt?

LÖSUNG

Setze 5 parallele Review-Agents ein, jeder spezialisiert auf ein Thema:


Bug Hunter


Security


Style


Tests


Docs

Jeder Agent läuft parallel bei PR-Erstellung, postet Findings als Kommentare und eskaliert nur signifikante Issues an menschliche Reviewer.

[Diagramm: PR → 5 parallele Agents → Zusammengeführte Findings → Human Review Gate]

```
# Slash-Command zum Auslösen des Reviews
/review-pr --parallel --agents="bugs,security,style,tests,docs"

# Beispiel Agent-Prompt (Security)
"Analysiere diesen Diff auf Sicherheitsprobleme: Hardcoded Secrets,
SQL Injection, XSS-Vektoren, unsichere Dependencies.
Output: JSON mit Severity, Zeilennummer, Empfehlung."
```

ROI Spart 30-40% Senior-Dev-Zeit → €2.400-3.200/Monat pro Senior (bei €120/Std.)

2 Context Loader mit CLAUDE.md

Problem: Context Switching killt Produktivität

Jedes Mal, wenn Claude eine neue Konversation startet, vergisst es deine Architektur, Namenskonventionen und Tech Stack. Du verschwendest die ersten 5 Minuten mit dem Wiedererklären desselben Kontexts.

LÖSUNG

Erstelle eine projekt-spezifische `CLAUDE.md`-Datei im Repo-Root. Claude Code lädt diese automatisch bei jeder Session und erhält so persistenten Kontext über dein Projekt.

```
mein-projekt/ ├── CLAUDE.md ← Auto-geladener Kontext ├── src/ ├── tests/ ├── package.json └── README.md
```

```
# CLAUDE.md

## Projektübersicht
E-Commerce API mit Node.js + TypeScript + PostgreSQL.

## Architektur
- src/controllers/ → Route Handler
- src/services/ → Business-Logik
- src/models/ → Prisma Schema

## Konventionen
- Kebab-case für Dateien, camelCase für Variablen
- Alle Endpoints geben { data, error, meta } zurück
- Tests nutzen Vitest; ausführen mit `pnpm test`

## Befehle
- `pnpm dev` → Dev-Server starten (Port 3000)
- `pnpm db:migrate` → Migrationen ausführen
```

Pro-Tipp

Füge team-spezifische Prompts hinzu wie "Prüfe immer auf N+1 Queries" oder "Nutze unseren Custom Logger, nicht console.log". Je spezifischer, desto besser.

ROI Neue Mitarbeiter produktiv in Tagen, nicht Wochen. Null Kontext-Wiederholung.

3 Standards Enforcer Hooks

Problem: Inkonsistente Code-Styles

"Kannst du diese Variable umbenennen?" "Dieses Pattern nutzen wir hier nicht." Die Hälfte deiner PR-Kommentare sind Style-Debatten, keine Logik-Diskussionen.

LÖSUNG

Pre-commit Hooks, die Claude-gestützte Checks ausführen, bevor Code den PR erreicht. Probleme werden beim Schreiben gefangen, nicht beim Review.


```
// .husky/pre-commit
#!/bin/sh

# Claude Standards-Check auf gestagete Dateien
claude-code check-standards --staged \
  --rules="naming,imports,error-handling" \
  --fix-minor \
  --fail-on-major

# Exit-Codes:
# 0 = Alles gut (oder Minor Issues auto-gefixt)
# 1 = Major Issues gefunden, Commit blockiert
```

```
# .claude/standards.yaml
naming:
  files: kebab-case
  variables: camelCase
  constants: SCREAMING_SNAKE_CASE

imports:
  order: [builtin, external, internal, relative]
  no-default-export: true

error-handling:
  require-try-catch-in: [controllers, services]
  custom-error-class: AppError
```

ROI Reduziert PR-Style-Debatten um 50%. Konsistente Codebase = schnellere Reviews.

4 Auto-Documentation Agent

Problem: Veraltete Dokumentation

Dein README sagt "Führe `npm start` aus", aber ihr seid vor sechs Monaten auf pnpm gewechselt. API-Docs sind drei Versionen veraltet. Niemand will Docs schreiben.

LÖSUNG

Merge-getriggerter Agent, der automatisch Dokumentation aktualisiert, wenn sich Code ändert. Überwacht strukturelle Änderungen und hält Docs synchron.

[Diagramm: PR Merged → Webhook → Claude Doc Agent → Aktualisierte README/API Docs → Auto-commit]

```
# .github/workflows/auto-docs.yml
name: Dokumentation aktualisieren
on:
  push:
    branches: [main]
    paths:
      - 'src/**'
      - 'package.json'

jobs:
  update-docs:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Claude Doc Agent ausführen
        run: |
          claude-code agent run doc-updater \
            --scope="README.md,docs/api.md" \
            --context="Änderungen in diesem Commit" \
            --auto-commit
```

```
# Doc Agent Prompt-Template
"Prüfe die Code-Änderungen in diesem Commit. Aktualisiere Dokumentation:
1. README.md - Befehle, Setup-Anweisungen, Dependencies
2. docs/api.md - Endpoint-Signaturen, Request/Response-Shapes
3. CHANGELOG.md - Eintrag für diese Änderung hinzufügen

Regeln: Bestehende Struktur beibehalten. Nur aktualisieren, was sich geändert hat.
Unsichere Updates mit [REVIEW NEEDED] markieren."
```

ROI Hält Wissen aktuell. Spart 10+ Stunden/Monat bei manuellen Doc-Updates.

5 Project Sync mit MCP-Integrationen

Problem: Tool-Wechsel (Jira/Slack/GitHub)

"Lass mich mal das Ticket checken..." Tab-Wechsel. "Ich poste in Slack..." Tab-Wechsel. "Was war die PR-Nummer?" Tab-Wechsel. Context Switching zerstört deinen Flow.

LÖSUNG

MCP (Model Context Protocol) Server, die Claude direkt mit deinen Tools verbinden. Fragen stellen, Tickets aktualisieren und Nachrichten senden, ohne den Editor zu verlassen.


```
// claude_config.json - MCP Server Setup
{
  "mcpServers": {
    "jira": {
      "command": "npx",
      "args": ["@anthropic/mcp-server-jira"],
      "env": {
        "JIRA_URL": "https://deinteam.atlassian.net",
        "JIRA_TOKEN": "${JIRA_API_TOKEN}"
      }
    },
    "slack": {
      "command": "npx",
      "args": ["@anthropic/mcp-server-slack"],
      "env": {
        "SLACK_TOKEN": "${SLACK_BOT_TOKEN}"
      }
    }
  }
}
```

```
# Jetzt kannst du Claude fragen:
"Was ist der Status von PROJ-123?"
"Poste eine Zusammenfassung meiner Änderungen in #engineering"
"Erstelle ein Ticket für diesen Bug, den ich gerade gefunden habe"
```

ROI Reduziert Context Switches um 60%. Länger im Flow-Zustand bleiben.

6 Security Auditor Agent

Problem: Schwachstellen in KI-generiertem Code

KI generiert Code schnell. Manchmal zu schnell. Hardcoded API Keys, SQL Injection-Vektoren und XSS-Schwachstellen rutschen durch, wenn man schnell unterwegs ist.

LÖSUNG

Dedizierter Security Agent, der jede KI-generierte Änderung auf häufige Schwachstellen scannt, bevor sie in Produktion geht.

[Diagramm: Code-Änderung → Security Agent Scan → Risk Report → Gate: Pass/Block/Review]

```
# Security Audit Prompt-Template
"Scanne diesen Code auf Sicherheitslücken:

CHECKLISTE:
□ Hardcoded Secrets (API Keys, Passwörter, Tokens)
□ SQL Injection-Vektoren (String-Konkatenation in Queries)
□ XSS-Schwachstellen (unsanitierter User-Input in HTML)
□ Path Traversal (User-Input in Dateipfaden)
□ Unsichere Dependencies (bekannte CVEs)
□ Fehlende Authentication/Authorization Checks
□ Sensitive Data Exposure (PII in Logs, Error Messages)

OUTPUT-FORMAT:
{
  "severity": "critical|high|medium|low",
  "location": "datei:zeile",
  "issue": "beschreibung",
  "fix": "empfehlung"
}

Falls keine Issues: { "status": "clean", "confidence": 0.0-1.0 }"
```

⚠ Wichtig

Verlasse dich nie ausschließlich auf KI für Security. Nutze dies als First-Pass-Filter, nicht als Ersatz für ordentliche Security Audits und Penetration Testing.

ROI Verhindert kostspielige Breaches. Compliance-Boost für SOC2/ISO Audits.

7 Multi-Feature Execution Loop

Problem: Sequentielle Planung limitiert parallele Arbeit

Du hast 5 Features für diesen Sprint zu liefern. Aber Claude kann nur an einer Sache gleichzeitig arbeiten… oder doch?

LÖSUNG

Ein Custom Claude Skill, der ein Planungsdokument in parallele Ausführung transformiert. Es zerlegt deine Roadmap in unabhängige Tasks, weist Sub-Agents zu und loopt bis zur Fertigstellung — ermöglicht 3-5 Features parallel.

So funktioniert es

1. **Input:** Lade ein Planungsdokument hoch (Feature Specs, Roadmap)
2. **Analyse:** Claude zerlegt die Arbeit in unabhängige Tasks
3. **Delegation:** Sub-Agents werden jedem Task zugewiesen
4. **Execution Loop:** Tasks laufen parallel, mit Iteration bei Fehlern
5. **Output:** Zusammengeführte Code-Branches bereit für Review

```
# Execution Loop Skill Prompt
"Analysiere diesen Plan: [PLAN_DOCUMENT]

PROZESS:
1. In unabhängige Tasks aufteilen (keine Cross-Dependencies)
2. Für jeden Task einen Sub-Agent erstellen:
   - /code-gen für Implementierung
   - /test für Test-Coverage
   - /doc für Dokumentation
3. Alle Tasks parallel ausführen
4. Bei Fehler: Retry mit modifiziertem Ansatz (max 3 Versuche)
5. Bei Erfolg: Commit auf Feature Branch
6. Loop bis alle Tasks abgeschlossen

OUTPUT: Zusammenfassung der erledigten Arbeit + Branch-Namen zum Mergen"
```

Git Worktree für parallele Entwicklung

Kombiniere dies mit Git Worktrees — jedes Feature bekommt seinen eigenen isolierten Checkout:

```
main (primärer Worktree)
    ../feature1-worktree → feature1-branch
```

```
../feature2-worktree → feature2-branch
../feature3-worktree → feature3-branch
```

```
# Worktrees für parallele Features aufsetzen
git worktree add ../feature1-wt feature1-branch
git worktree add ../feature2-wt feature2-branch

# Jeder Claude Agent arbeitet in seinem eigenen Worktree
# Kein Stashing, kein Branch-Wechsel, keine Konflikte

# Mergen wenn fertig
git merge feature1-branch
git worktree remove ../feature1-wt
```

Build #7 Fortsetzung: Sicherheit & Setup

⚠ **Sicherheits-Constraints**

Guardrails, die du MUSST implementieren:

- Token/Kosten-Caps in Prompts (verhindert unkontrollierte Ausgaben)
- Error-Handling Loops mit max. Retry-Limits
- Human-Review Gates vor jedem Merge in main
- CI-Checks, die kaputte Builds blockieren

Sicherheitsüberlegungen:

- Enterprise Claude Accounts mit Datenisolation nutzen
- Niemals sensitiven Code (Secrets, PII) in Prompts
- Integration mit sicherem MCP nur für interne Tools
- Audit-Logs für alle Agent-Aktionen

Setup-Schritte:

1. Claude Code Plugin für Agents installieren
2. Git Worktrees im Repo konfigurieren
3. Loop zuerst an Dummy-Feature testen
4. Monitoring via Logs (Output zu Slack/Discord)

Limits

- Max 5 parallele Tasks (mehr = abnehmender Ertrag + Überlastung)
- Immer `git worktree prune` nach Merges
- CI-Checks auf allen agent-generierten Code erzwingen
- Alle Merges reviewen — KI ist ein Multiplikator, kein Ersatz

ROI **3-5 Features parallel bearbeiten. Spart Wochen pro Sprint-Zyklus.**

Alles zusammenbringen

Implementierungsreihenfolge

#2 CLAUDE.md → #3 Hooks → #1 Review → #6 Security → #5 MCP → #7 Loops → #4 Docs

Starte mit #2 (CLAUDE.md) — es ist kostenlos, dauert 30 Minuten und verbessert sofort jede Claude-Interaktion.

Best Practices:

- Plugins und Prompts versionieren
- Dokumentieren, was jeder Agent tut, für das Team
- Trainings-Sessions beim Rollout neuer Setups
- Vorher/Nachher-Metriken messen, um ROI zu beweisen

Skaliere Claude Code für dein Team

Du hast jetzt 7 praxiserprobte Setups, um Claude Code vom individuellen Produktivitäts-Hack zur team-weiten Superkraft zu transformieren.

€5.000 - €15.000

Geschätzte monatliche Einsparungen (5-Personen Engineering-Team)

Was diese Setups ermöglichen

- **Compound-Effekte** — Jedes Setup macht die anderen effektiver
- **Konsistente Qualität** — Standards automatisch durchgesetzt, nicht durch Debatten
- **Schnelleres Shipping** — Parallele Arbeit + automatisierte Reviews = kürzere Zyklen
- **Geringeres Risiko** — Security-Checks fangen Probleme vor Produktion ab
- **Besseres Onboarding** — Neue Mitarbeiter produktiv in Tagen mit CLAUDE.md

Erhalte individuelle ROI-Projektionen für dein Team

Das €997 Team-Audit enthält:

- 90-minütiger Deep-Dive in euren aktuellen Workflow
- Priorisierte Setup-Empfehlungen für euren Stack
- Individuelle ROI-Berechnungen basierend auf Teamgröße
- Implementierungs-Roadmap mit Zeitplan

Team-Audit buchen →

Bonus

Antworte auf deine Download-Email für eine **kostenlose 15-minütige Plugin-Beratung**. Ich beantworte deine spezifischen Fragen zur Implementierung dieser Setups.

---

codecoast labs GmbH

julian@codecoast.ch | @codecoast