AIDER

Aider

Die komplette Anleitung

Stand: Juli 2026

KI-gestütztes Pair Programming im Terminal

Autor Christian Drapatz

Plattform Terminal · macOS · Linux · Windows

Version 1.0

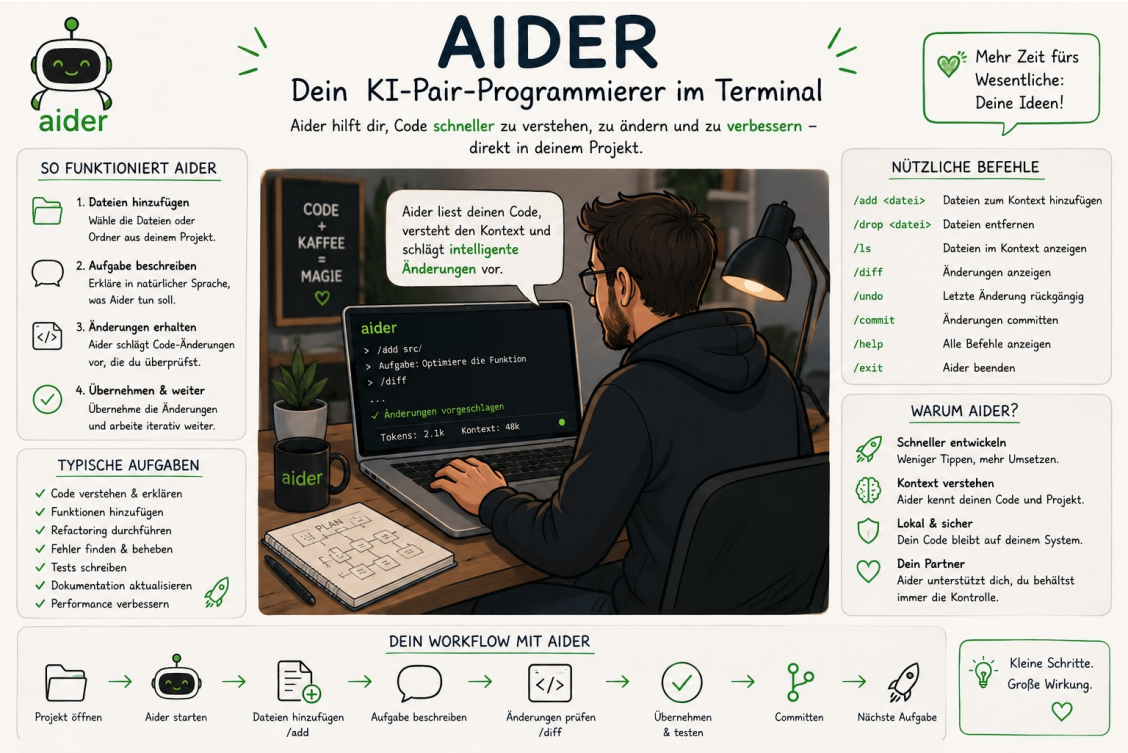
Inhaltsverzeichnis

Wird beim Öffnen in Word automatisch aktualisiert. Falls nicht: Rechtsklick auf diesen Text → Felder aktualisieren.

Stand: Juli 2026

Was ist Aider?

Aider ist ein Open-Source-Kommandozeilen-Tool für **KI-gestütztes Pair Programming**. Es läuft im Terminal, liest den bestehenden Code eines Projekts und baut daraus eine Repo-Map. Relevante Dateien schickt es als Kontext an ein LLM deiner Wahl. Änderungen wendet Aider als Diffs an und committet sie automatisch mit einer generierten Commit-Message. Git ist die Source of Truth: Jede Änderung lässt sich nachvollziehen und rückgängig machen.



Aider Workflow

Homepage: https://aider.chat

Wem gehört Aider?

Aider wurde 2023 von Paul Gauthier als Nebenprojekt gestartet und ist unter der Apache-2.0-Lizenz frei verfügbar (Repository `Aider-AI/aider`, ursprünglich `paul-gauthier/aider`). Ein Unternehmen steckt nicht dahinter, es ist ein reines Community-Projekt – mittlerweile mit über 40.000 GitHub-Stars und einer wachsenden Zahl an Mitwirkenden. Releases erscheinen etwa alle zwei Wochen.

Zukunft von Aider

Aider wird aktiv weiterentwickelt, unter anderem in Richtung:

- Erweiterte Modellunterstützung (z. B. neue Gemini- und OpenAI-Modelle)
- Zusätzliche Sprachunterstützung (u. a. Swift über tree-sitter-Grammatiken, MATLAB, Clojure)
- Bessere Cloud/Local-LLM-Flexibilität
- Performance-Verbesserungen bei großen Repositories

Der größte Vorteil bleibt die Modell-Unabhängigkeit. Während andere Tools an ein bestimmtes LLM-Ökosystem gebunden sind, kann Aider jederzeit den Anbieter wechseln.

Installation

Voraussetzung ist Python (empfohlen: Python 3.9 bis 3.12) sowie Git. Dieser Weg über Homebrew und pipx ist getestet und läuft zuverlässig unter macOS.

1. Homebrew installieren, falls noch nicht vorhanden:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
brew --version
```

Bei Bedarf Homebrew aktualisieren:

```
brew update
brew upgrade
```

2. Python installieren, falls noch nicht vorhanden:

```
brew install python
```

3. pipx installieren und zum PATH hinzufügen:

```
brew install pipx
pipx ensurepath
```

Danach das Terminal schließen und neu öffnen, damit der angepasste PATH aktiv wird.

4. Aider installieren:

```
pipx install aider-chat
```

Aider läuft laut Dokumentation stabil mit Python 3.9 bis 3.12. Mit neueren Versionen wie 3.13 kommt es gelegentlich zu Problemen. Tritt das auf, hilft eine gezielt installierte unterstützte Version:

```
brew install python@3.12
pipx uninstall aider-chat
pipx install aider-chat --python python3.12
```

5. Installation prüfen:

```
aider --version
# oder
pipx list
```

6. Aider im eigenen Projekt starten:

```
cd /Pfad/zu/deinem/Projekt
aider
```

Beim allerersten Start in einem Repository fragt Aider, ob es seine eigenen Verlaufsdateien automatisch zur `.gitignore` hinzufügen darf:

```
Add .aider* to .gitignore (recommended)? (Y)es/(N)o [Yes]:
```

Mit Yes bestätigen. Welche `.aider*`-Dateien das betrifft, steht im Kapitel "Wichtige Dateien" weiter unten. Mit `/help` direkt im Chat lassen sich anschließend alle verfügbaren Befehle anzeigen.

7. Aktualisieren und Deinstallieren:

```
pipx upgrade aider-chat     # nur Aider aktualisieren
pipx upgrade-all            # alle über pipx installierten Tools
aktualisieren
pipx uninstall aider-chat   # Aider deinstallieren
```

Kurzstart mit Ollama (lokales Modell)

Wer direkt lokal statt mit einem Cloud-Anbieter starten möchte, prüft zunächst die bereits heruntergeladenen Ollama-Modelle:

```
ollama list
```

Beispielausgabe:

```
NAME                       ID              SIZE      MODIFIED
```

```
gemma3:4b                    a2af6cc3eb7f    3.3 GB    3 weeks ago
nomic-embed-text:latest      0a109f422b47    274 MB    3 weeks ago
llama3.2:3b                  a80c4f17acd5    2.0 GB    4 weeks ago
```

Anschließend den Ollama-Dienst starten und Aider damit verbinden:

```
ollama serve
export OLLAMA_API_BASE=http://localhost:11434
aider --model ollama_chat/qwen3-coder:30b
```

oder mit einem anderen lokalen Modell, z. B.:

```
aider --model ollama_chat/gemma3
```

Die ausführlichen Hintergründe zu Ollama selbst – Installation, Modellverwaltung, unterstützte LLMs – stehen im Kapitel "Ollama" weiter unten. Die Kombination mit Xcode ist im Kapitel "Aider mit lokalem LLM nutzen und in Xcode einbinden" beschrieben.

Warum sollte man Aider verwenden?

- Git-Integration: Jede Änderung landet als sauberer, atomarer Commit
- Funktioniert mit praktisch jedem LLM, ohne Bindung an einen Anbieter
- Hohe Token-Effizienz, dadurch günstiger im Betrieb als viele Alternativen
- Reifes Projekt, seit 2023 im produktiven Einsatz, mit großer Community
- Editor-unabhängig: läuft neben jedem beliebigen Editor

Einbindung in den Entwicklerprozess

Aider lässt sich flexibel in bestehende Workflows einbauen:

- Als zusätzliches Terminal-Fenster oder Tmux-Split neben dem Editor der Wahl
- Im "Watch-Files"-Modus: Aider beobachtet Dateien und reagiert auf spezielle KI-Kommentare, die man direkt im Editor in den Code schreibt
- Im Architect-Modus: ein Modell plant die Änderung, ein zweites (günstigeres) Modell setzt sie um
- In CI/Skripten über die Kommandozeile für automatisierte Refactoring-Aufgaben
- Über Git-Hooks, um Aider gezielt in bestimmten Projektphasen einzubinden

Unterschied zu Claude Code, OpenCode & Co.

Kriterium	Aider	Claude Code	OpenCode
Modellbindung	Beliebiges LLM	Nur Claude-Modelle	75+ Provider über Models.dev
Sprache/ Implementierung	Python	– (Anthropic-Produkt)	Go
Git-Workflow	Auto-Commit je Änderung	Manuell/optional	Manuell/optional
Reifegrad	Sehr hoch, seit 2023	Neuer, aber sehr poliert	Neuer, community-getrieben
Fokus	Reine Editier-Engine im Terminal	Vollständiger Agenten-Workflow mit tiefer Anthropic-Integration	Offenheit und Provider-Flexibilität

Aider ist die "chirurgische" Lösung für gezielte Code-Änderungen mit sauberer Git-Historie. Claude Code punktet bei komplexen, mehrstufigen Agenten-Aufgaben und tiefer Anthropic-Integration. OpenCode bietet die größte Provider-Auswahl.

Vorteile und Nachteile

Vorteile:

- Keine Vendor-Lock-in-Situation, freie Modellwahl
- Sehr gute Git-Nachvollziehbarkeit
- Kostengünstig durch hohe Token-Effizienz
- Läuft überall, wo Python läuft
- Großes, aktives Community-Wissen (Issues, Doku, Foren)

Nachteile:

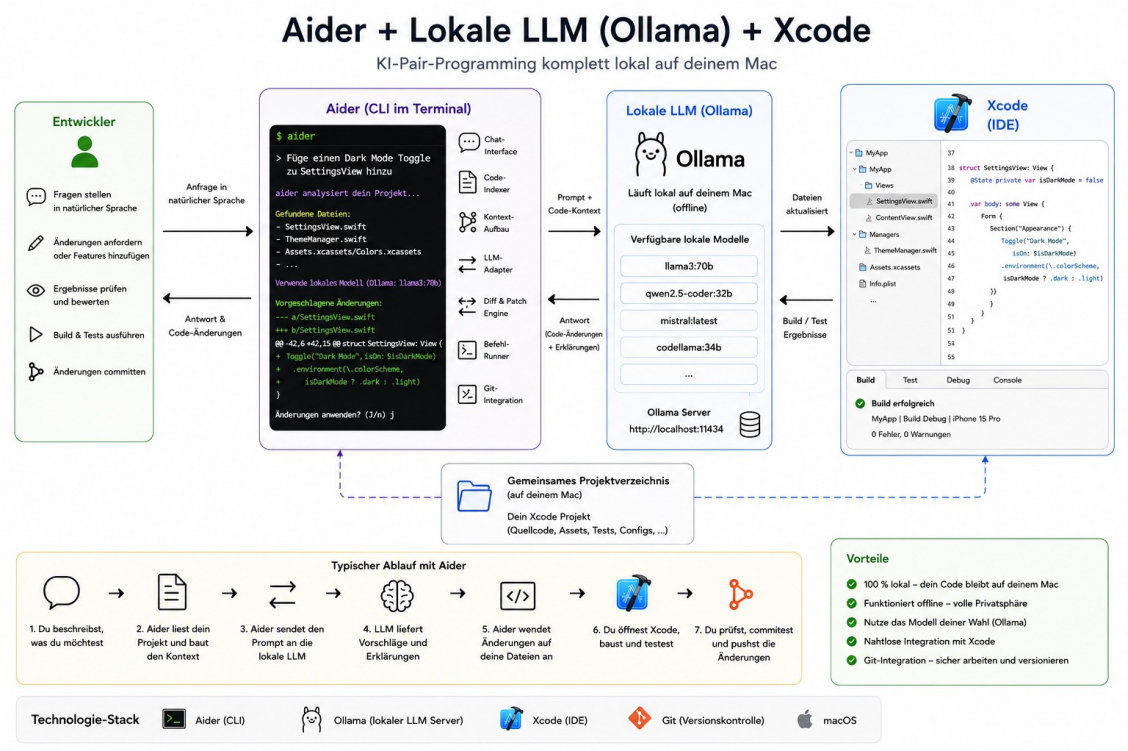
- Kein offizieller Support durch einen Konzern, reines Community-Projekt
- Kein natives IDE-Plugin – Integration erfolgt indirekt über Dateisystem/Terminal
- Bei sehr komplexen, mehrstufigen Agenten-Aufgaben tendenziell schwächer als spezialisierte Agenten-Tools
- Swift/Xcode-Unterstützung noch nicht so ausgereift wie bei etablierteren Sprachen

Einbindung in Xcode

Aider hat kein natives Xcode-Plugin. Die Integration läuft indirekt:

1. Aider im Terminal (z. B. in einem separaten Fenster oder iTerm2-Split) im Projektordner starten
2. Xcode bleibt parallel geöffnet – Aider ändert Dateien direkt auf der Festplatte
3. Xcode erkennt externe Änderungen automatisch und lädt die Dateien neu
4. Mit dem Watch-Modus (`aider --watch-files`) kann man direkt im Xcode-Editor Kommentare wie `// aider: refactor this to use async/await` schreiben, die Aider dann automatisch aufgreift und umsetzt

Die Codebase-Analyse (Repo-Map) läuft bei Swift-Projekten über tree-sitter-Grammatiken. Diese Unterstützung wird laufend verbessert, ist aber im Vergleich zu Python oder JavaScript noch jünger.

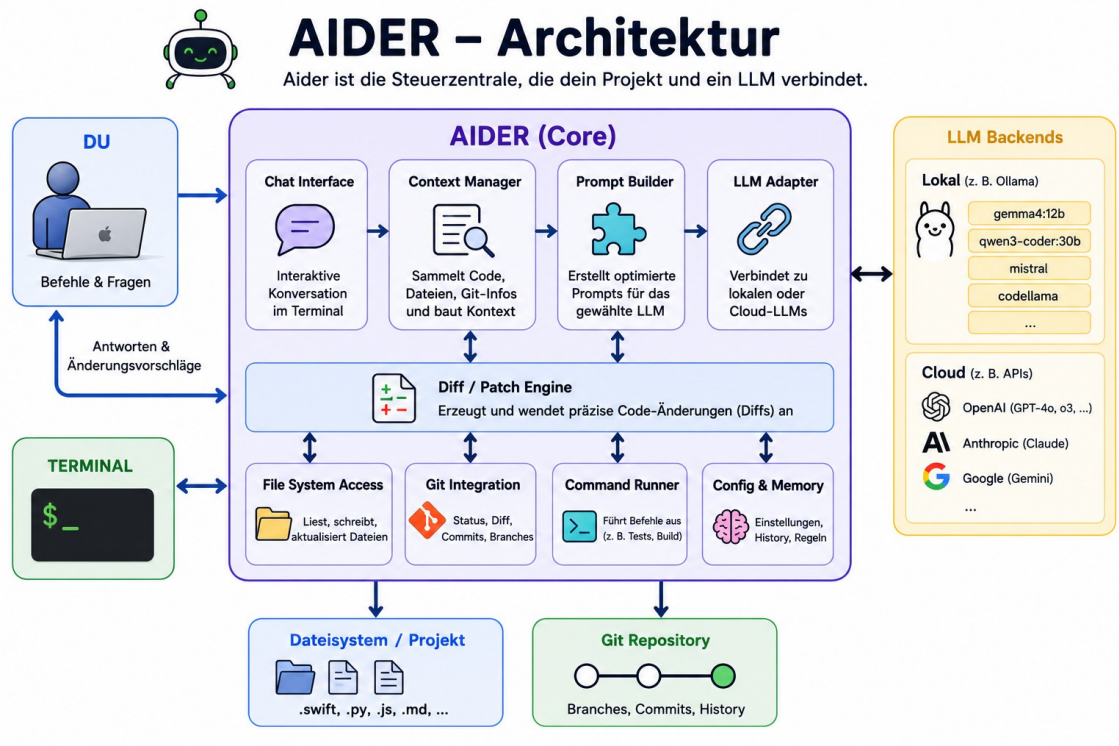


Einbindung in Xcode

Architektur

Aider besteht im Kern aus folgenden Bausteinen:

- Repo-Map: Ein kompakter, tree-sitter-basierter Überblick über Symbole und Struktur des Projekts, der auch bei großen Codebasen in den Kontext passt
- Prompt-/Context-Manager: Wählt relevante Dateien und Ausschnitte für die jeweilige Anfrage aus
- Edit-Format-Engine: Wendet die vom LLM zurückgegebenen Änderungen als Diffs bzw. Suchen-und-Ersetzen-Blöcke auf die echten Dateien an
- Git-Layer: Committet jede erfolgreiche Änderung automatisch mit generierter Commit-Message
- Modell-Adapter-Schicht: Abstrahiert die Anbindung an verschiedene LLM-Provider über eine einheitliche Schnittstelle (u. a. via LiteLLM)
- Watch-Mode: Ein Dateisystem-Beobachter, der auf Kommentar-Trigger in beliebigen Editoren reagiert



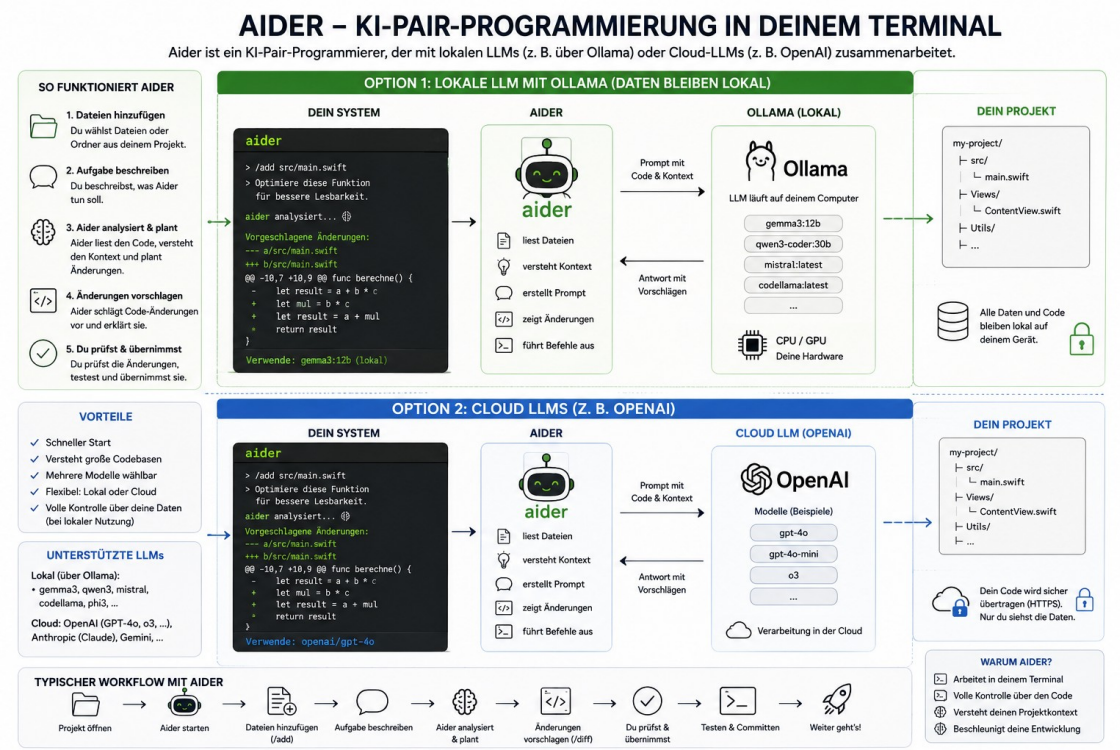
Aider Architektur

Unterstützte LLM-Modelle

Aider ist explizit modell-agnostisch und unterstützt unter anderem:

- Anthropic Claude (Sonnet, Opus, Haiku)
- OpenAI GPT-4o, o1, o3-mini, o3-pro
- Google Gemini
- DeepSeek R1 und DeepSeek Chat V3
- Lokale Modelle über Ollama
- Praktisch jeden weiteren Anbieter mit OpenAI-kompatibler API

Meist lohnt sich eine Kombination: ein starkes Architect-Modell (z. B. Claude Sonnet) für die Planung, ein günstigeres Editor-Modell für die reine Codeänderung.



Unterstützte LLM-Modelle

Benötigt man zusätzliche Tools?

Für den Basisbetrieb reicht:

- Python 3.9 oder neuer
- Git
- Ein API-Key für mindestens ein unterstütztes LLM (oder ein lokal laufendes Modell via Ollama)

Aider läuft laut Dokumentation stabil mit Python 3.9 bis 3.12. Mit neueren Versionen wie 3.13 kann es zu Problemen kommen. Im Zweifel eine unterstützte Version gezielt installieren, wie im Kapitel "Installation" beschrieben.

Optional, aber empfehlenswert:

- pipx zur sauberen Installation in einer isolierten Umgebung
- Ein Terminal-Multiplexer wie tmux oder ein Split-Terminal (z. B. in iTerm2), um Aider parallel zum Editor laufen zu lassen
- swift-format als Lint-/Formatierungstool, wenn man mit Swift-Projekten arbeitet

Workflow mit Aider – wie läuft das ab?

Ein typischer Aider-Durchlauf sieht so aus:

1. Im Projektordner (Git-Repository) `aider` starten
2. Relevante Dateien mit `/add` in den Chat-Kontext holen, damit Aider sie bearbeiten darf
3. In natürlicher Sprache beschreiben, was geändert werden soll
4. Aider schlägt einen Diff vor, wendet ihn direkt auf die Dateien an und erstellt automatisch einen Git-Commit mit passender Commit-Message
5. Ergebnis im Editor (z. B. Xcode) oder per `git diff` prüfen, bei Bedarf mit `/undo` den letzten Commit zurückrollen
6. Nächste Anweisung geben oder Chat mit `/clear` zurücksetzen

Für größere Aufgaben empfiehlt sich ein zweistufiger Ablauf: Erst im Architect-Modus planen lassen, danach die konkrete Umsetzung im Code-Modus durchführen lassen. So trennt man Planung (kann ein teureres, stärkeres Modell übernehmen) von der reinen Editierarbeit (günstigeres Modell reicht oft aus).

Im Alltag hat sich dieser Zyklus als Faustformel bewährt, egal ob mit Cloud- oder lokalem Modell:

Aufgabe → Änderungen → `/diff` → Build → Testen → Git-Commit → nächste Aufgabe

Zwei ausführlich durchgespielte Beispiele – ein kleines Kommandozeilenprojekt sowie ein täglicher SwiftUI-Zyklus mit lokalem Ollama-Modell – stehen im Kapitel "Praxisbeispiele: Schritt-für-Schritt an echten Projekten" weiter unten.

Alle wichtigen Chat-Befehle

Ja, Aider hat ein umfangreiches Set an Slash-Befehlen, ähnlich wie Claude Code. Sie lassen sich grob in folgende Gruppen einteilen:

Datei-Verwaltung:

Befehl	Funktion
`/add <datei>`	Datei(en) zum bearbeitbaren Kontext hinzufügen (Wildcards möglich)
`/read-only <datei>`	Datei nur als Referenz laden bzw. bestehende Datei auf read-only setzen
`/drop <datei>`	Datei wieder aus dem Kontext entfernen

Befehl	Funktion
`/ls`	Alle bekannten Dateien und ihren Chat-Status auflisten
`/map`	Aktuelle Repo-Map anzeigen
`/map-refresh`	Repo-Map neu erzeugen

Modus wechseln:

Befehl	Funktion
`/code`	Normaler Editier-Modus (Standard)
`/architect`	Architect-Modus: Planung durch ein Modell, Umsetzung durch ein zweites
`/ask`	Nur Fragen stellen, ohne dass Code verändert wird
`/context`	Kontext-Modus, um den umgebenden Code einzugrenzen
`/chat-mode <modus>`	Aktiven Modus dauerhaft (sticky) umschalten

Modell & Einstellungen:

Befehl	Funktion
`/model <name>`	Hauptmodell für die laufende Sitzung wechseln
`/models <suchbegriff>`	Verfügbare Modelle durchsuchen
`/editor-model <name>`	Editor-Modell (im Architect-Modus) wechseln
`/weak-model <name>`	Modell für einfache Hilfsaufgaben (z. B. Commit-Messages) wechseln
`/reasoning-effort <wert>`	Reasoning-Aufwand setzen (z. B. low/medium/high)
`/settings`	Aktuelle Konfiguration anzeigen
`/tokens`	Token-Verbrauch der aktuellen Sitzung anzeigen

Git, Historie & Sitzung:

Befehl	Funktion
`/diff`	Änderungen seit der letzten Nachricht anzeigen
`/commit`	Aktuellen Stand manuell committen
`/undo`	Letzten Aider-Commit zurückrollen
`/git <befehl>`	Beliebigen Git-Befehl ausführen
`/clear`	Chat-Verlauf zurücksetzen
`/reset`	Alle Dateien entfernen und Chat-Verlauf zurücksetzen
`/save <datei>`	Aktuelles Datei-Setup der Sitzung in eine Datei speichern
`/load <datei>`	Gespeicherte Befehle aus einer Datei laden und ausführen

Ausführen, Testen & Prüfen:

Befehl	Funktion
`/run <befehl>` (Alias `!`)	Shell-Befehl ausführen, Ausgabe optional in den Chat übernehmen
`/test <befehl>`	Befehl ausführen, Ausgabe bei Fehlschlag automatisch in den Chat übernehmen
`/lint`	Konfigurierten Linter laufen lassen und Ergebnisse einbeziehen

Sonstiges:

Befehl	Funktion
`/voice`	Spracheingabe statt Tippen nutzen
`/paste`	Text oder Bild aus der Zwischenablage einfügen
`/editor` (Alias `/edit`)	Externen Editor zum Schreiben der nächsten Nachricht öffnen

Befehl	Funktion
`/web <url>`	Webseite abrufen, in Markdown wandeln und in den Chat übernehmen
`/copy`	Letzte Antwort in die Zwischenablage kopieren
`/copy-context`	Aktuellen Chat-Kontext als Markdown exportieren (z. B. für Web-UIs)
`/report`	Problem als GitHub-Issue melden
`/help`	Hilfe zu Aider anzeigen
`/exit` bzw. `/quit`	Aider beenden

Gibt es Äquivalente zu /init, /usage, /status aus Claude Code?

Teilweise. Einige Claude-Code-Befehle haben eine direkte Entsprechung bei Aider, andere fehlen bewusst, weil das Konzept dahinter nicht passt:

- `/clear` gibt es bei Aider genauso, mit identischer Funktion
- `/model` gibt es ebenfalls, ergänzt um `/models` zum Durchsuchen verfügbarer Modelle sowie `/editor-model` und `/weak-model`, da Aider bis zu drei verschiedene Modellrollen gleichzeitig verwenden kann
- `/usage` in Claude Code zeigt das Verbrauchs-/Abo-Kontingent an. Ein Äquivalent gibt es bei Aider nicht, da man dort meist mit einem eigenen API-Key direkt beim Anbieter bezahlt. Am nächsten kommt `/tokens`, das den Token-Verbrauch der aktuellen Chat-Sitzung anzeigt
- `/status` in Claude Code fasst Sitzungsinformationen zusammen. Bei Aider übernehmen das zusammen `/settings` (aktuelle Konfiguration) und `/ls` (welche Dateien geladen und editierbar sind)
- `/init` in Claude Code analysiert die Codebase automatisch und erzeugt daraus eine CLAUDE.md. Ein Äquivalent fehlt bei Aider: Die `CONVENTIONS.md` (siehe oben) muss man selbst anlegen und pflegen, es gibt keinen Befehl, der das Projekt automatisch analysiert und daraus Regeln ableitet

Wichtige Dateien: Gibt es ein Äquivalent zu .claude und CLAUDE.md?

Ein einzelnes, zentrales Verzeichnis wie `.claude` mit Unterordnern für Settings, Befehle, Agents und Hooks gibt es bei Aider nicht. Stattdessen verteilt sich die Konfiguration auf mehrere einzelne Punktdateien, die üblicherweise im Projektstamm oder im Home-Verzeichnis liegen:

Datei	Entspricht bei Claude Code	Zweck
`.aider.conf.yml`	`settings.json`	Zentrale Verhaltenskonfiguration: Standardmodell, Auto-Commits, Lint-/Testbefehle, welche Dateien automatisch geladen werden usw. Kann im Home-Verzeichnis (global) und im Repo-Root (projektspezifisch) gleichzeitig existieren, wobei sich die Werte überlagern
`CONVENTIONS.md` (frei benennbar)	`CLAUDE.md`	Coding-Regeln und Projektkonventionen, die bei jeder Anfrage mitgeschickt werden, wenn man sie per `--read` oder im Config-File einträgt
`.aiderignore`	am ehesten `.gitignore` bzw. Permission-Deny-Regeln	Legt fest, welche Dateien/Ordner nie automatisch in den Chat-Kontext geladen werden, z. B. Secrets oder generierte Dateien
`.env`	`.env` bzw. Umgebungsvariablen in `settings.json`	API-Keys und Modell-Umgebungsvariablen, sollte nie versioniert werden
`.aider.model.settings.yml`	kein direktes Äquivalent	Erlaubt es, Einstellungen für einzelne Modelle zu überschreiben oder zu ergänzen, etwa welches Edit-Format ein bestimmtes Modell nutzen soll
`.aider.model.metadata`	kein direktes Äquivalent	Hinterlegt Kontextfenstergröße

Datei	Entspricht bei Claude Code	Zweck
`.json`		und Kosten für Modelle, die Aider von sich aus noch nicht kennt – wichtig z. B. bei neuen oder selbst gehosteten Ollama-Modellen
`.aider.chat.history.md`	Session-Transkripte	Wird automatisch geführt und enthält den vollständigen Chatverlauf aller Sitzungen im Markdown-Format – praktisch als Nachvollziehbarkeits- und Audit-Log
`.aider.input.history`	Shell-/Eingabeverlauf	Speichert die zuletzt eingegebenen Prompts, damit man mit der Pfeiltaste nach oben auch sitzungsübergreifend frühere Eingaben wiederfinden kann

Die Verlaufsdateien `.aider.chat.history.md` und `.aider.input.history` sowie die generierten Metadaten-Dateien gehören in die `.gitignore`, es sind reine Laufzeitdaten. `.aider.conf.yml`, `CONVENTIONS.md` und `.aiderignore` dagegen sollten ins Repository, damit alle im Team dieselbe Grundkonfiguration nutzen.

Muss man wirklich jede relevante Datei mit /add hinzufügen?

Nicht zwingend, aber mit Einschränkungen. Wichtig ist der Unterschied zwischen "Aider kennt eine Datei" und "Aider darf eine Datei bearbeiten":

- Die Repo-Map läuft immer im Hintergrund mit. Sie enthält für das ganze Projekt eine komprimierte Übersicht aus Datei-, Klassen- und Funktionssignaturen, auch für Dateien, die nicht per `/add` geladen wurden. Dadurch kennt Aider grob Struktur und Inhalt des Projekts
- Bearbeiten darf Aider trotzdem nur Dateien, die im Chat explizit als editierbar hinzugefügt wurden. Braucht Aider eine weitere Datei, fragt es im interaktiven Modus nach: "Aider möchte Datei X zum Chat hinzufügen, erlauben? (yes/no/always)". Mit "always" oder dem Start-Flag `--yes-always` lässt sich diese Nachfrage dauerhaft abschalten

- Für kleinere, gezielte Änderungen lohnt es sich trotzdem, die betroffenen Dateien selbst per `/add` zu benennen. Zu viel ungefragt geladener Kontext lenkt das Modell ab und treibt die Token-Kosten unnötig hoch (siehe Kapitel zu den Kosten)

Für die drei genannten Fälle heißt das konkret:

Neue Datei erzeugen lassen: Am zuverlässigsten gibt man den noch nicht existierenden Dateinamen selbst per `/add Sources/NewFile.swift` an, bevor man den Auftrag beschreibt. Aider legt die Datei dann beim Anwenden des Diffs automatisch an. Verlässt man sich stattdessen darauf, dass Aider von sich aus eine neue Datei erstellt, landen Änderungen in der Praxis gelegentlich in einer bereits geladenen, bestehenden Datei – das Modell hängt stark am vorhandenen Kontext.

"Finde den Fehler": Ohne hinzugefügte Dateien sieht Aider nur die komprimierte Repo-Map, nicht den vollständigen Quellcode. Für eine offene Fehlersuche reicht das allein oft nicht. Besser: zunächst selbst oder per `/run <testbefehl>` die Fehlermeldung bzw. den Stacktrace erzeugen und in den Chat geben, dazu die Dateien, die laut Stacktrace oder eigener Vermutung betroffen sind. Aider fragt bei Bedarf automatisch nach weiteren Dateien, die es aufgrund der Repo-Map oder des Fehlerbilds für relevant hält. Der `/ask`-Modus eignet sich gut für eine erste Analyse ohne Editierrisiko, bevor man in den Code-Modus wechselt.

"Das Feature muss neu eingebaut werden" (großer, unscharfer Scope): Hier lohnt sich der Architect-Modus (`/architect`). Man beschreibt das gewünschte Feature, Aider erarbeitet zunächst einen Plan und nennt dabei meist selbst, welche Dateien betroffen sind oder neu entstehen müssen. Auf dieser Basis lassen sich die passenden Dateien gezielt hinzufügen, statt vorab zu raten. Alternativ helfen Wildcards wie `/add Sources/Feature/**/*.swift`, um einen ganzen Modulordner auf einmal freizugeben, wenn der Umfang noch unklar ist. Der größere Kontext kostet dann allerdings auch mehr Token.

Gibt es so etwas wie ".claude", Rules oder Agents?

Aider hat kein Konzept von "Subagenten" wie Claude Code, aber es gibt vergleichbare Mechanismen für Konfiguration und Regeln:

- `.aider.conf.yml` im Projekt- oder Home-Verzeichnis: steuert, wie Aider läuft (Standardmodell, automatische Commits ja/nein, welche Dateien automatisch als read-only geladen werden usw.), vergleichbar mit `settings.json` bei Claude Code
- `CONVENTIONS.md`: eine frei benennbare Datei mit Coding-Regeln und Projektkonventionen, die bei jeder Anfrage automatisch mitgeschickt wird, wenn man sie per `--read CONVENTIONS.md` oder im Config-File einträgt – entspricht inhaltlich einer `CLAUDE.md`. Sie sollte unter ca. 200 Zeilen bleiben, damit die Regeln zuverlässig befolgt werden
- `.aiderignore`: analog zu `.gitignore`, legt fest, welche Dateien Aider nie automatisch einbeziehen soll

- `.env`-Datei: für API-Keys und Modell-Umgebungsvariablen
- Architect-/Editor-Modell-Kombination: kommt dem "Planungs-Agent + Ausführungs-Agent"-Prinzip am nächsten, ist aber fest in zwei Rollen aufgeteilt statt frei definierbarer Subagenten

Ein direktes Äquivalent zu einem Agenten-Ökosystem mit beliebig vielen spezialisierten Subagenten gibt es also nicht. Die Kombination aus `.aider.conf.yml` (Verhalten) und `CONVENTIONS.md` (Regeln) deckt aber ab, was man von einer Projektkonfiguration erwartet.

Ollama

Was ist Ollama?

Ollama ist ein Open-Source-Tool, um große Sprachmodelle lokal auf dem eigenen Rechner herunterzuladen, zu verwalten und laufen zu lassen, vergleichbar mit "Docker für KI-Modelle". Im Hintergrund nutzt es die llama.cpp-Inferenz-Engine und kapselt Quantisierung, Speicherverwaltung und GPU-Beschleunigung hinter einer einfachen Kommandozeile und einer lokalen REST-API. Modelle speichert Ollama wie eine Container-Registry als wiederverwendbare Layer und Blobs.

Installation

macOS, Linux und Windows werden unterstützt.

macOS:

```
brew install ollama
```

oder das Installer-Paket von ollama.com herunterladen. Linux:

```
curl -fsSL https://ollama.com/install.sh | sh
```

Nach der Installation läuft Ollama als Hintergrunddienst und ist standardmäßig unter `http://127.0.0.1:11434` erreichbar.

Workflow mit Ollama

1. Modell herunterladen: `ollama pull qwen2.5-coder:14b`
2. Modell interaktiv testen: `ollama run qwen2.5-coder:14b`
3. Laufende Modelle anzeigen: `ollama ps`
4. Modell wieder entfernen: `ollama rm qwen2.5-coder:14b`
5. Über die REST-API (`/api/chat`, `/api/generate`) von anderen Tools ansprechen lassen – genau darüber bindet sich später auch Aider an

Welche LLMs unterstützt Ollama?

Ollama bietet eine große Modellbibliothek unter ollama.com/library, u. a.:

- Llama 3.x / Llama-Familie (Meta)
- Qwen2.5-Coder – speziell für Coding-Aufgaben empfohlen
- DeepSeek-Coder / DeepSeek R1 (destillierte, lokal lauffähige Varianten)
- Mistral / Mixtral
- Gemma (Google)
- Phi (Microsoft)

Modelle werden nach dem Schema `name:tag` referenziert, wobei der Tag Parametergröße und Quantisierung codiert, z. B. `llama3.1:8b-q4_K_M`. Mit Quantisierung laufen bereits leistungsfähige 7B/8B-Modelle auf Laptops mit 8–16 GB RAM; größere Modelle (14B, 32B, 70B) brauchen entsprechend mehr RAM bzw. eine GPU.

Aider mit lokalem LLM nutzen und in Xcode einbinden

Schritt 1: Ollama vorbereiten

```
ollama pull qwen2.5-coder:14b
```

Ollama muss im Hintergrund laufen (nach der Installation ist das automatisch der Fall, sonst mit `ollama serve` starten).

Schritt 2: Aider mit dem lokalen Modell verbinden

Über Umgebungsvariablen (empfohlen):

```
export OLLAMA_API_BASE=http://127.0.0.1:11434
aider --model ollama_chat/qwen2.5-coder:14b
```

Der Präfix `ollama_chat/` sorgt dafür, dass Aider den Chat-Endpunkt (`/api/chat`) von Ollama nutzt, statt des einfacheren Generate-Endpunkts.

Alternativ dauerhaft in `~/.aider.conf.yml` hinterlegen:

```
model: ollama_chat/qwen2.5-coder:14b
```

So muss man das Modell nicht bei jedem Start erneut angeben.

Lokale Modelle sind meist schwächer als Cloud-Modelle wie Claude oder GPT-4o, deshalb lohnt sich oft eine Kombination im Architect-Modus: ein Cloud-Modell übernimmt die Planung, ein lokales Ollama-Modell die reine Umsetzung. Wer Datenschutz priorisiert, dreht die Rollen um und bleibt komplett lokal.

Schritt 3: Einbindung in Xcode

Da Aider kein natives Xcode-Plugin hat, läuft die Einbindung wie im Xcode-Kapitel oben beschrieben rein über das Dateisystem:

1. Xcode-Projekt öffnen

2. In einem separaten Terminal-Fenster oder -Split (z. B. iTerm2 neben Xcode) in den Projektordner wechseln und Aider mit dem lokalen Modell starten
3. Relevante Swift-Dateien per `/add` hinzufügen
4. Änderungen anfordern – Aider schreibt sie direkt in die `.swift`-Dateien auf der Festplatte
5. Xcode erkennt die externen Änderungen automatisch und aktualisiert die geöffneten Editor-Tabs
6. Optional den Watch-Modus nutzen: `aider --model ollama_chat/qwen2.5-coder:14b --watch-files` – dann reicht es, direkt im Xcode-Editor einen Kommentar wie `// aider: refactor to async/await` zu schreiben, den Aider im Hintergrund aufgreift und automatisch umsetzt
7. Build und Tests weiterhin ganz normal in Xcode ausführen; Aider committet parallel jede Änderung in Git, sodass jederzeit ein sauberer Rücksprungpunkt existiert

Vorteil dieser Kombination: Modell-Inferenz, Code-Änderung und Build laufen komplett lokal, es geht kein Code an einen Cloud-Anbieter. Nachteil: Lokale Modelle liefern bei komplexen Swift- und SwiftUI-Aufgaben meist schwächere Ergebnisse als aktuelle Cloud-Modelle wie Claude Sonnet. Diese Variante eignet sich deshalb vor allem für einfachere, repetitive Änderungen oder für Projekte mit hohen Datenschutzanforderungen.

Praxisbeispiele: Schritt-für-Schritt an echten Projekten

Beispiel 1: Neue Methode und Unit-Test in einem Swift-Kommandozeilenprojekt

Als Beispiel dient ein kleines Swift-Kommandozeilenprojekt mit einer Datei `Calculator.swift`:

```
struct Calculator {
    func add(_ a: Int, _ b: Int) -> Int {
        a + b
    }
}
```

Ablauf:

1. Terminal im Projektordner öffnen und Aider starten: `aider --model sonnet`
2. Datei in den Kontext holen: `/add Sources/Calculator.swift`
3. Auftrag formulieren: "Füge eine Methode subtract hinzu, die zwei Int-Werte subtrahiert, und schreibe dazu einen XCTest in CalculatorTests.swift."
4. Aider antwortet mit einer kurzen Erklärung des geplanten Vorgehens und zeigt anschließend einen Diff, z. B. die neue Methode `func subtract(_ a: Int, _ b: Int) -> Int { a - b }` in `Calculator.swift` sowie eine neue Datei `Tests/CalculatorTests.swift` mit einem passenden `XCTestCase`
5. Nach Bestätigung wendet Aider den Diff an und erstellt automatisch einen Commit, z. B. mit der Message `Add subtract method to Calculator with unit test`
6. Mit `/run swift test` lässt sich der neue Test direkt aus dem Aider-Chat heraus ausführen; schlägt er fehl, schickt Aider die Fehlermeldung automatisch zurück ins Modell und schlägt eine Korrektur vor
7. Ergebnis in Xcode ansehen: Die Datei wurde extern verändert, Xcode lädt sie automatisch neu, der neue Test taucht im Test-Navigator auf
8. Mit `/diff` lässt sich der letzte Commit noch einmal nachvollziehen, mit `/undo` bei Bedarf rückgängig machen

Datei hinzufügen, Auftrag formulieren, Diff prüfen, committen, testen: Dieser kurze Zyklus ist der Kern jedes Aider-Workflows, egal ob im Swift-, Python- oder JavaScript-Projekt.

Beispiel 2: Täglicher Arbeitszyklus in einem SwiftUI-Projekt mit lokalem Modell

Im Alltag eines bestehenden SwiftUI-Projekts läuft der gleiche Zyklus meist so ab, hier mit einem lokalen Ollama-Modell statt eines Cloud-Modells:

1. Projekt öffnen

```
aider ––model ollama_chat/gemma4:12b
```

2. Dateien hinzufügen

```
/add Sources/SettingsView.swift
```

3. Eine Aufgabe geben

```
Optimiere die Performance dieser View.
```

4. Änderungen prüfen

```
/diff
```

5. In Xcode testen und bauen

6. Wenn alles passt, committen

```
/commit
```

Danach beginnt der nächste Zyklus mit einer neuen Aufgabe. Für SwiftUI-Projekte hat sich dieser Zyklus bewährt:

Aufgabe → Änderungen → `/diff` → Xcode-Build → Testen → Git-Commit → nächste Aufgabe

Beispiele für typische Aufgaben, die man Aider einfach als normalen Chat-Text gibt, ganz ohne speziellen Befehl:

- "Analysiere die SettingsView und vereinfache den Code, ohne das Verhalten zu ändern."
- "Suche nach möglichen Memory Leaks."
- "Ersetze alle veralteten SwiftUI-APIs durch aktuelle."
- "Schreibe Unit-Tests für den LocationManager."

Aider liest dann die Dateien, die zuvor per `/add` hinzugefügt wurden, und versucht, die Aufgabe damit umzusetzen.

Kosten bei Cloud-LLMs am Beispiel OpenAI

Aider selbst ist kostenlos, es fallen aber Kosten beim jeweiligen Modell-Anbieter an, abhängig von Input- und Output-Token. Am Beispiel der OpenAI-API (Stand 2026):

Modell	Input je 1 Mio. Token	Output je 1 Mio. Token	Einsatzbereich
GPT-5	1,25 $	10 $	Starkes Allround-Modell, gutes Preis-Leistungs-Verhältnis
GPT-5 Mini	0,25 $	2 $	Günstiges Editor-Modell für den Architect-Modus
GPT-4o	2,50 $	10 $	Legacy-Preis für Bestandsnutzer
GPT-4o mini	0,15 $	0,60 $	Sehr günstig für einfache Änderungen

Für den Alltag mit Aider relevant:

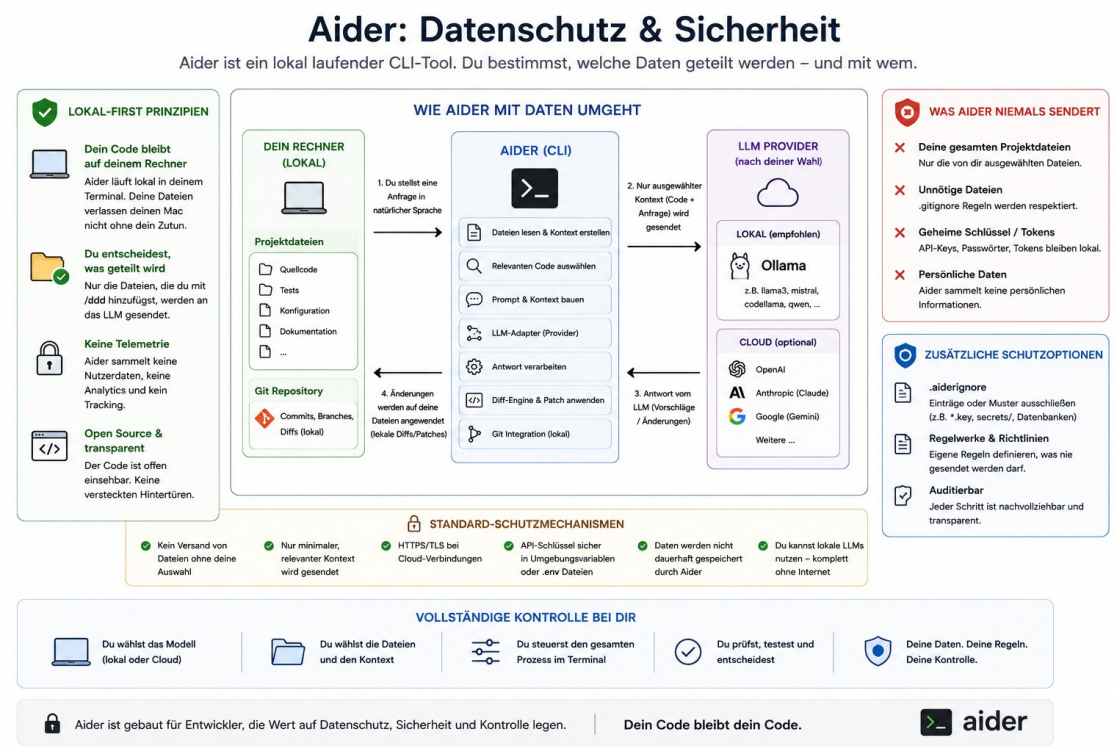
- Die Repo-Map und alle im Chat befindlichen Dateien zählen bei jeder Anfrage als Input-Token. Je mehr Dateien per `/add` geladen sind, desto teurer wird jede Nachricht
- Prompt-Caching (bei OpenAI und Anthropic verfügbar) senkt wiederholte Input-Kosten um bis zu 90 Prozent, wenn derselbe Kontext mehrfach hintereinander läuft. Deshalb lohnt es sich, `CONVENTIONS.md` und selten wechselnde Dateien mit `/read` statt `/add` zu laden
- Mit `/tokens` im Aider-Chat lässt sich der aktuelle Verbrauch pro Nachricht direkt einsehen
- Ein zweistufiges Setup (starkes Architect-Modell nur für die Planung, günstiges Editor-Modell für die eigentliche Umsetzung) senkt die Gesamtkosten oft deutlich gegenüber der Nutzung eines einzigen Spitzenmodells für jede Anfrage
- Als kostenlose Alternative steht, wie oben beschrieben, jederzeit ein lokales Ollama-Modell zur Verfügung

Datenschutz und Sicherheit

Sobald Aider mit einem Cloud-Modell arbeitet, schickt es alle in den Chat geladenen Dateien sowie die generierte Repo-Map an den jeweiligen Anbieter (z. B. OpenAI, Anthropic, Google). Bei sensiblem Code, Kundendaten oder Geschäftsgeheimnissen sollte man sich dessen bewusst sein:

- Anbieter unterscheiden sich darin, ob und wie lange Anfragen für Trainingszwecke gespeichert werden – die jeweiligen API-Nutzungsbedingungen prüfen, gerade bei Firmenprojekten
- Nur die Dateien mit `/add` in den Chat holen, die für die aktuelle Aufgabe wirklich nötig sind, statt das gesamte Repository auf einmal freizugeben
- Für Projekte mit hohen Datenschutzanforderungen (z. B. Gesundheits- oder Finanzdaten) bietet sich die vollständig lokale Variante mit Ollama an, da dabei kein Code das eigene Netzwerk verlässt
- Von Aider erzeugte Commits sollten wie jeder andere Commit einem Code-Review unterzogen werden, bevor sie in geschützte Branches gelangen

Zur Telemetrie: Aider selbst sammelt standardmäßig keine Nutzungsdaten. Anonyme Analytics sind laut offizieller Datenschutzerklärung eine reine Opt-in-Funktion. Beim ersten Start fragt Aider explizit nach Zustimmung, ohne Zustimmung bleibt die Erfassung dauerhaft deaktiviert. Laut eigener Aussage erfasst Aider dabei nie Code, Chat-Inhalte oder API-Keys, sondern höchstens anonymisierte Nutzungsereignisse unter einer zufälligen UUID. Das betrifft ausschließlich Aider selbst. Bei Nutzung eines Cloud-Modells gehen Code und Prompts weiterhin an den jeweiligen LLM-Anbieter, wie oben beschrieben.



Datenschutz und Sicherheit

Umgang mit Secrets und API-Keys

- API-Keys niemals im Klartext in `.aider.conf.yml` oder direkt im Prompt hinterlegen, wenn diese Dateien versioniert werden
- Stattdessen eine `.env`-Datei im Projektverzeichnis anlegen (Aider liest sie automatisch ein) und diese in `.gitignore` eintragen
- Für dauerhafte, rechnerweite Konfiguration Umgebungsvariablen im Shell-Profil setzen, z. B. `export ANTHROPIC_API_KEY=…` oder `export OPENAI_API_KEY=…`
- Bei Teams empfiehlt sich ein Secret-Manager (z. B. 1Password CLI, macOS Keychain oder ein Vault-Dienst), aus dem die Umgebungsvariablen beim Start der Shell geladen werden, statt Keys manuell weiterzugeben
- Regelmäßig prüfen, ob versehentlich Keys in Commits gelandet sind, z. B. mit `git log -p | grep -i api_key` oder einem Secret-Scanning-Tool

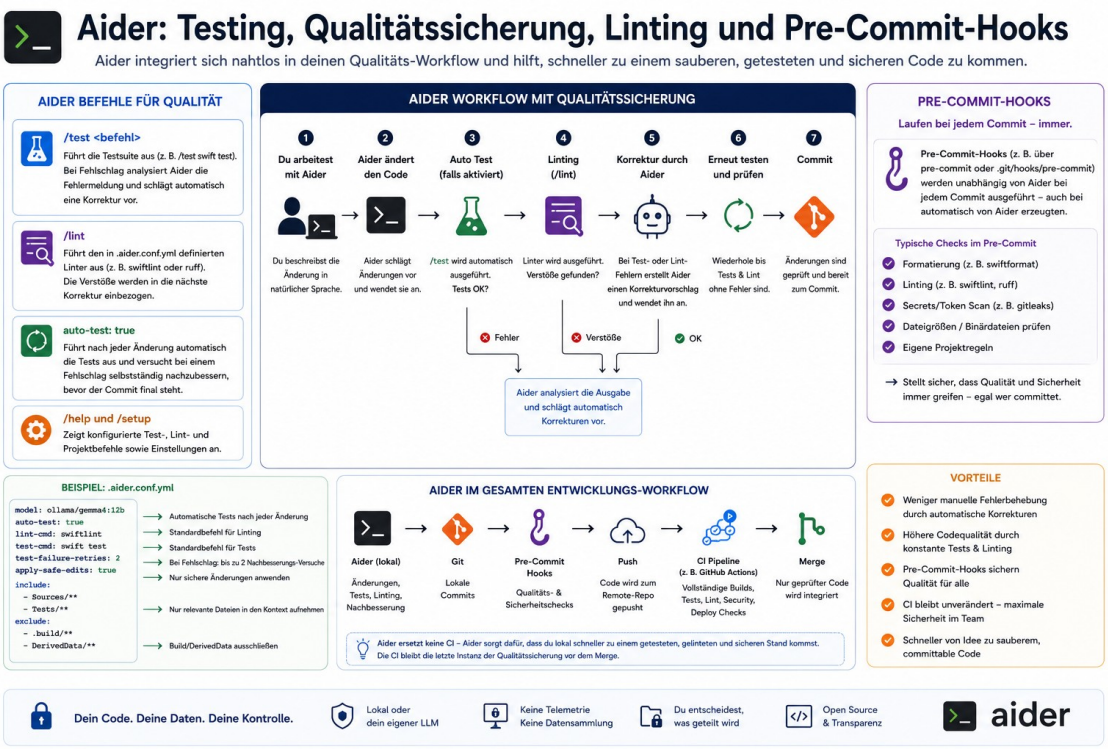
.aiderignore – sensible Dateien ausschließen

Analog zu `.gitignore` legt eine `.aiderignore`-Datei im Projektstamm fest, welche Dateien und Ordner Aider grundsätzlich ignoriert, unabhängig davon, ob sie im Git-Repository liegen. Typischer Inhalt:

```
.env
*.pem
*.p12
Secrets/
*.xcconfig
GoogleService-Info.plist
node_modules/
```

So landen Zertifikate, Provisioning-Profile, API-Keys oder generierte Konfigurationsdateien nie versehentlich im Chat-Kontext oder gar bei einem Cloud-Modell, auch nicht, wenn jemand aus Versehen `/add *` verwendet.

Testing, Qualitätssicherung, Linting und Pre-Commit-Hooks



Testing, Qualitätssicherung, Linting und Pre-Commit-Hooks

Aider lässt sich gut in bestehende Qualitätssicherungs-Workflows einbinden:

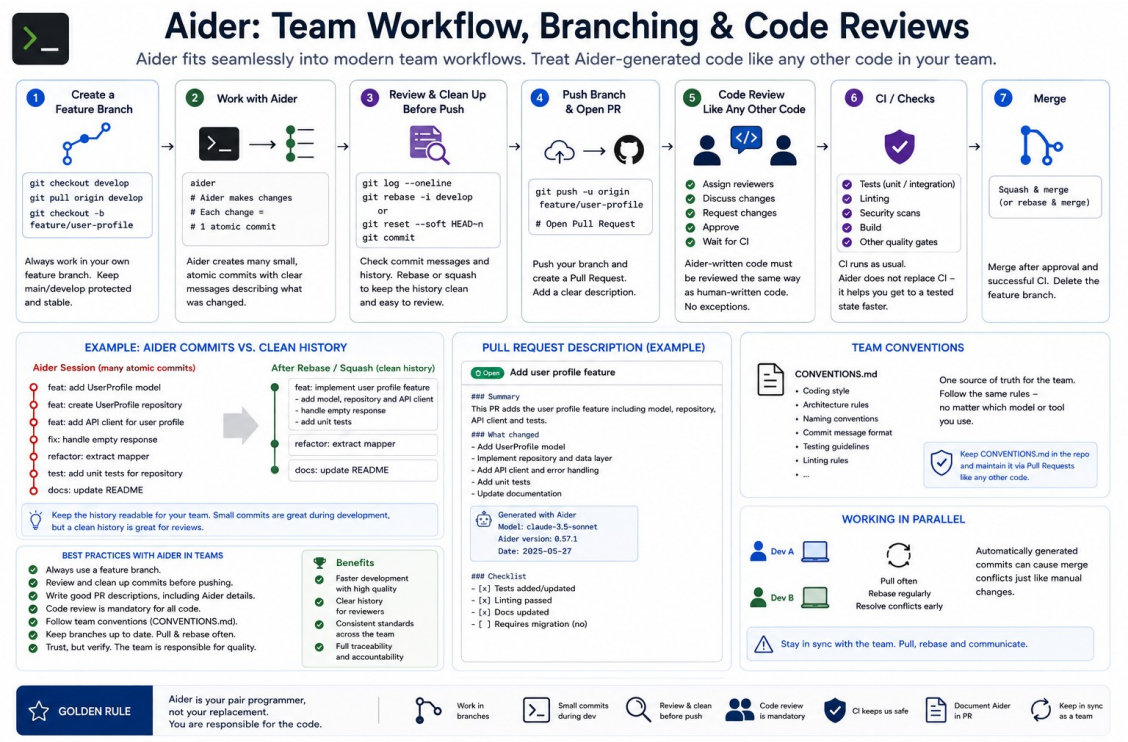
- Mit `/test <befehl>` (z. B. `/test swift test` oder `/test pytest`) lässt Aider die Testsuite laufen und bei einem Fehlschlag automatisch einen Korrekturvorschlag erarbeiten, basierend auf der tatsächlichen Fehlermeldung
- Mit `/lint` lässt sich ein in `.aider.conf.yml` hinterlegter Linter (z. B. `swiftlint` oder `ruff`) ausführen; Aider bezieht die gemeldeten Verstöße direkt in die nächste Korrektur mit ein
- In `.aider.conf.yml` können Standardbefehle für Tests und Linting hinterlegt werden, sodass sie nicht bei jeder Sitzung neu angegeben werden müssen:

```
lint-cmd: swiftlint
test-cmd: swift test
auto-test: true
```

- Mit `auto-test: true` führt Aider nach jeder Änderung automatisch die Tests aus und versucht bei einem Fehlschlag selbstständig nachzubessern, bevor der Commit final steht

- Pre-Commit-Hooks (z. B. über das Python-Tool `pre-commit` oder ein Git-Hook-Skript unter `.git/hooks/pre-commit`) laufen unabhängig von Aider bei jedem Commit, auch bei den automatisch erzeugten. So greifen Formatierung, Linting oder ein Secret-Scan immer, egal ob ein Mensch oder Aider committet
- Empfehlenswert: CI-Pipeline (z. B. GitHub Actions) unverändert weiterlaufen lassen – Aider ersetzt keine CI, sondern beschleunigt nur den Weg bis zum committeten, lokal getesteten Stand

Team-Workflow: mehrere Entwickler, Branching und Code-Reviews



Team-Workflow

Da Aider jede Änderung als eigenen Git-Commit ablegt, lässt es sich gut in gängige Team-Workflows integrieren:

- Jede Aider-Sitzung sollte in einem eigenen Feature-Branch stattfinden, genau wie bei manueller Entwicklung – so bleibt `main` bzw. `develop` unberührt
- Da Aider häufig viele kleine, atomare Commits erzeugt, empfiehlt sich vor dem Merge ein interaktives Rebase oder Squash, um die Historie für Reviewer übersichtlich zu halten
- Commit-Messages von Aider sind automatisch generiert und beschreiben meist präzise, was geändert wurde – trotzdem lohnt sich vor dem Push eine kurze manuelle Durchsicht mit `git log`, um sicherzustellen, dass die Botschaften zum Teamstandard passen
- Code-Reviews sollten Aider-generierte Commits genauso behandeln wie von Menschen geschriebene: Pull Request erstellen, Reviewer zuweisen, CI abwarten. Der Umstand, dass eine KI den Code geschrieben hat, befreit nicht von der Review-Pflicht
- Um Nachvollziehbarkeit zu schaffen, hat es sich in Teams bewährt, in der PR-Beschreibung zu vermerken, dass Aider (und mit welchem Modell) beteiligt war
- Eine gemeinsame `CONVENTIONS.md` im Repository sorgt dafür, dass alle Teammitglieder unabhängig vom verwendeten Modell nach denselben Coding-Regeln arbeiten – sie sollte wie normaler Code über Pull Requests gepflegt werden

- Bei paralleler Arbeit mehrerer Entwickler mit Aider im selben Repository empfiehlt sich wie immer häufiges Pullen/Rebasen, da automatisch erzeugte Commits ebenso zu Merge-Konflikten führen können wie manuelle Änderungen

Sprachmodus (Voice)

Mit dem Befehl `/voice` lässt sich Aider per Spracheingabe statt über die Tastatur bedienen. Aider nimmt über das Mikrofon eine Sprachnachricht auf, wandelt sie über ein Spracherkennungsmodell in Text um und verwendet diesen Text als nächste Chat-Anfrage, genau so, als hätte man ihn getippt. Praktisch vor allem, um längere, frei formulierte Anforderungen schnell zu diktieren, ohne den Gedankenfluss durchs Tippen zu unterbrechen. Vorausgesetzt wird ein konfigurierter Zugang zu einem Spracherkennungsdienst, je nach Konfiguration etwa die OpenAI-Whisper-API. Dafür kann ein zusätzlicher API-Key nötig sein, falls nicht ohnehin schon ein OpenAI-Key hinterlegt ist.

Bilder und Screenshots im Chat verwenden (Multimodal)

Aider kann bei vision-fähigen Modellen (z. B. GPT-4o oder Claude Sonnet) auch Bilder direkt im Chat verarbeiten, nicht nur Text. Für UI-Arbeit an SwiftUI-Views ist das nützlich. Zwei Wege, ein Bild hinzuzufügen:

- Per Kommandozeile beim Start: `aider screenshot.png` fügt das Bild direkt als Kontext hinzu, zusätzlich zu den sonstigen Optionen
- Während einer laufenden Sitzung mit `/paste`, um ein Bild direkt aus der Zwischenablage einzufügen

Typische Einsatzszenarien:

- Einen Screenshot eines UI-Fehlers in Xcode/im Simulator machen und Aider bitten, die Ursache in der zugehörigen SwiftUI-View zu finden und zu beheben
- Ein Mockup oder Design-Entwurf (z. B. aus Figma exportiert) einfügen und Aider daraus eine erste SwiftUI-Implementierung erstellen lassen
- Eine schwer per Text zu übertragende Fehlermeldung oder ein Crash-Log als Screenshot statt als Copy-Paste-Text einfügen

Nicht jedes Modell kann Bilder verarbeiten. Bei lokalen Ollama-Modellen ist Vision-Unterstützung die Ausnahme, dafür braucht es gezielt ein Vision-Modell wie `llava`. Die gängigen Cloud-Modelle wie GPT-4o oder Claude Sonnet unterstützen Bilder dagegen standardmäßig.

Vergleich mit weiteren Tools: Cursor, GitHub Copilot, Windsurf

Neben Claude Code und OpenCode lohnt sich auch die Abgrenzung zu den verbreiteten IDE-zentrierten Tools:

Tool	Ansatz	Xcode-Bezug	Besonderheit
Aider	Terminal, editor-agnostisch	Indirekt über Dateisystem/Watch-Modus	Freie Modellwahl, saubere Git-Historie
Cursor	Eigenständige IDE (VS-Code-Fork)	Kein Xcode, eigene IDE nötig	Sehr tiefe Codebase-Analyse, hohe Popularität
GitHub Copilot	Editor-Erweiterung	Offizielle Extension direkt in Xcode verfügbar	Breiteste Editor-Unterstützung, große Verbreitung in Firmen
Windsurf	Eigenständige IDE mit Cascade-Agent	Kein Xcode, eigene IDE nötig	Stark agentisch, iteriert selbstständig bis zum fertigen Ergebnis

Für Xcode-Nutzer ist das ein wichtiger Unterschied: GitHub Copilot bringt eine offizielle Xcode-Extension mit und fügt sich damit am nahtlosesten in die native Oberfläche ein. Aider erreicht ein ähnliches Ergebnis nur indirekt über den Watch-Modus und externe Dateiänderungen, bietet dafür aber freie Modellwahl und keine Bindung an einen bestimmten Anbieter. Cursor und Windsurf verlangen dagegen den kompletten Umstieg auf eine neue IDE und stehen für native Xcode-/Swift-Projekte nicht direkt zur Verfügung.

Wichtige Sicherheitshinweise und Best Practices

Aider ändert Dateien selbstständig, committet automatisch und kann auf Wunsch sogar Shell-Befehle ausführen. Grund genug für ein eigenes Kapitel zur Kontrolle des Ganzen.

Auto-Commit-Verhalten verstehen

Aider committet standardmäßig jede erfolgreich angewendete Änderung sofort als eigenen Git-Commit (`AIDER_AUTO_COMMITS`, Standard: aktiviert). Das hilft der Nachvollziehbarkeit, führt aber auch dazu, dass leicht mehrere Commits entstehen, bevor man den Code wirklich gegengelesen hat:

- Diffs vor dem Bestätigen bewusst lesen, nicht blind auf "yes" drücken, besonders bei sicherheitsrelevantem Code (Auth, Zahlungsabwicklung, Datenzugriff)
- Mit `--no-auto-commits` lässt sich das automatische Committen abschalten, wenn man lieber selbst und gesammelt committen will. Das Flag schaltet laut aktuellem Kenntnisstand aber auch das separate Committen bereits vorhandener, unabhängiger Änderungen ("dirty commits") mit ab – das Verhalten also vorher genau prüfen
- Immer in einem echten Git-Repository arbeiten und den `--no-git`-Modus meiden. Nur über Git lässt sich jede Aider-Änderung zuverlässig zurückrollen (`/undo`, `git revert`)
- Vor größeren, riskanten Aufträgen einen eigenen Branch oder zumindest einen sauberen, committeten Ausgangszustand sicherstellen, damit ein Rücksprung jederzeit möglich ist

Vorsicht bei ausgeführten Shell-Befehlen

Befehle wie `/run`, `auto-test` und `auto-lint` lassen Aider Shell-Befehle beziehungsweise deren Ausgabe direkt verarbeiten. Das ist komfortabel, birgt aber Risiken:

- Ein `/run`-Befehl läuft ungeprüft in der eigenen Shell mit den eigenen Rechten. Vorschläge des Modells (z. B. "führe dies zur Fehlerbehebung aus") vor der Ausführung immer selbst lesen und verstehen
- Bei `auto-test: true` führt Aider die konfigurierte Testsuite nach jeder Änderung automatisch aus; das ist unkritisch, solange der Testbefehl selbst vertrauenswürdig und fest in `.aider.conf.yml` hinterlegt ist, statt vom Modell frei vorgeschlagen zu werden
- Besondere Vorsicht gilt, wenn im Repository Fremdcode oder Abhängigkeiten von Dritten liegen: Enthält eine Datei manipulierte Kommentare oder versteckte Anweisungen, kann ein Modell theoretisch dazu verleitet werden, unerwünschte Befehle vorzuschlagen (indirekte Prompt-

Injection). Deshalb gilt: keinem vorgeschlagenen Shell-Befehl blind vertrauen, nur weil er von der KI kommt
- In CI-Umgebungen oder automatisierten Skripten (`aider --yes-always`) sollte man sich der Tragweite bewusst sein, da dort keine manuelle Bestätigung mehr stattfindet

Edit-Formate kennen (diff, whole, udiff)

Aider nutzt je nach Modell unterschiedliche Formate, um Änderungen zu übermitteln:

- `diff` (Editblock-Format): Das Modell liefert Suchen-und-Ersetzen-Blöcke, Standard bei vielen aktuellen Modellen wie GPT-4o
- `whole`: Das Modell schreibt die komplette Datei neu, Standard bei schwächeren Modellen wie GPT-3.5, da diese mit präzisen Diffs öfter Fehler machen
- `udiff`: Ein universelles Diff-Format, u. a. bei GPT-4 Turbo im Einsatz

Aider setzt das passende Format meist automatisch anhand des gewählten Modells, es lässt sich bei Bedarf aber manuell über `--edit-format` erzwingen. Relevant wird das vor allem bei Fehlermeldungen wie "SEARCH/REPLACE block failed to match" – dann hilft oft ein Wechsel auf `whole` oder ein stärkeres Modell, statt den Fehler zu ignorieren.

Aider-Leaderboards

Unter aider.chat/docs/leaderboards veröffentlicht das Aider-Team regelmäßig aktualisierte Benchmarks, mit denen sich die Eignung verschiedener LLMs für Coding-Aufgaben vergleichen lässt. Zentral ist der sogenannte Polyglot-Benchmark: 225 anspruchsvolle Exercism-Aufgaben über sechs Sprachen (u. a. C++, Go, Java, JavaScript, Python, Rust), bei denen das Modell nicht nur isoliert Code erzeugt, sondern innerhalb der echten Aider-Edit-Schleife bewertet wird – inklusive einem zweiten Versuch, falls der erste Testlauf fehlschlägt.

Stand Juli 2026 führt GPT-5 (high) das Polyglot-Leaderboard mit rund 88 Prozent gelösten Aufgaben an, DeepSeek-V3.2-Exp ist das bestplatzierte frei verfügbare Open-Source-Modell. Vor der Wahl eines Modells für den produktiven Einsatz lohnt sich ein Blick auf das jeweils aktuelle Leaderboard, da sich die Rangfolge mit neuen Modellversionen häufig innerhalb weniger Wochen verschiebt.

Quellen

Diese Anleitung basiert neben eigenen, getesteten Praxisschritten auf einer Web-Recherche (Stand Juli 2026). Die wichtigsten verwendeten Quellen:

Offizielle Aider-Dokumentation und -Repository:

- Aider – offizielle Website
- Aider-Dokumentation
- In-Chat-Befehle
- Chat-Modi (Architect, Ask, Code)
- Coding-Konventionen (CONVENTIONS.md)
- Konfiguration allgemein
- YAML-Konfigurationsdatei
- Optionen-Referenz
- .env-Konfiguration
- Erweiterte Modell-Einstellungen
- Git-Integration
- Ollama-Anbindung
- Repository-Map
- Skripting/Automatisierung
- FAQ
- Aider-Leaderboards
- Release-Historie
- Bilder und Webseiten im Chat
- Analytics
- Datenschutzerklärung
- GitHub-Repository Aider-AI/aider
- GitHub Issue #2376 – Swift-Unterstützung

Ollama:

- Ollama API-Dokumentation
- Ollama GitHub-Repository

Modell-Preise:

- OpenAI API Pricing

Vergleiche und Hintergrundartikel (Community/Fachpresse, Stand 2026):

- Aider vs. OpenCode vs. Claude Code

- OpenCode vs. Claude Code vs. Aider – DEV Community
- Aider Cheat Sheet – ComputingForGeeks
- Aider Rules & Config Guide
- CONVENTIONS.md-Leitfaden
- Cursor vs. Windsurf vs. GitHub Copilot – CodeAnt
- Coding-Agenten-Vergleich – Artificial Analysis
- Agent Skills in Xcode – Hacking with Swift

> **Hinweis:** Da sich Preise, Modell-Rankings und Tool-Funktionsumfänge laufend ändern, sollten insbesondere die Kapitel zu Kosten, unterstützten Modellen und Leaderboards vor einer produktiven Entscheidung gegen die jeweils aktuelle offizielle Quelle geprüft werden.