Kimi Code – Die komplette Anleitung

Autor: Christian Drapatz

Stand: Juli 2026

Plattform: macOS · Linux · Windows

Version 1.0

Inhaltsverzeichnis

1 Einleitung

Kimi Code ist die aktuelle, in TypeScript/Node.js geschriebene Kommandozeilen-Anwendung von Moonshot AI für agentisches Programmieren. Sie läuft im Terminal, analysiert ein vorhandenes Projekt, liest und verändert Dateien, führt Shell-Befehle aus und kann Builds sowie Tests starten.

Dabei lohnt sich eine begriffliche Trennung, die in Marketingtexten oft verschwimmt: Kimi Code ist der **Coding-Agent**, also das Werkzeug, das mit dem Projekt interagiert, Freigaben einholt, Tools aufruft und den Arbeitsablauf steuert. Ein **Kimi-Modell** (zum Beispiel K3 oder K2.7 Code) ist dagegen nur das Sprachmodell, das die eigentlichen Textantworten und Tool-Aufrufe erzeugt. Kimi Code nutzt standardmäßig Kimi-Modelle, kann aber ebenso mit Modellen anderer Anbieter oder mit lokal betriebenen Modellen arbeiten. Agent und Modell sind also zwei austauschbare Ebenen, ganz ähnlich wie bei Claude Code, das ebenfalls unabhängig vom konkret gewählten Claude-Modell funktioniert.

Der Entwickler beschreibt das gewünschte Ergebnis in natürlicher Sprache. Kimi Code untersucht das Projekt, erstellt bei Bedarf einen Plan und führt die erforderlichen Schritte nach Bestätigung aus. Der Agent unterstützt unter anderem Skills, Hooks, MCP-Server, Plugins und Subagenten.

Kimi Code eignet sich vor allem für Aufgaben, bei denen der Agent mit einer vorhandenen Codebasis interagieren muss: Refactorings, Bugfixes, Migrationen, das Schreiben und Ausführen von Tests, Code-Reviews anhand von Projektregeln und projektweite Analysen. Für reine Chat- oder Wissensfragen ohne Projektbezug ist die allgemeine Kimi-Chat-Oberfläche der naheliegendere Einstiegspunkt, nicht die CLI.

2 Kimi Code und Claude Code im Vergleich

Kimi Code gehört zur gleichen Werkzeugkategorie wie Claude Code. Beide Agenten arbeiten direkt mit einem Projekt und verändern sowie testen Code selbstständig, statt ihn nur vorzuschlagen.

Bereich	Kimi Code	Claude Code
Anbieter	Moonshot AI	Anthropic
Bedienkonzept	Terminal-CLI, interaktiv oder mit `-p` nicht-interaktiv	Terminal-CLI, interaktiv oder mit `-p`/`--print` nicht-interaktiv
Plan-Modus	`/plan on` / `/plan off`, eigener Modus	`/plan`-Command bzw. Permission-Modus `plan`, auch per Tastenkombination umschaltbar
Datei- und Shell-Zugriff	Ja, über Tools mit Freigabepflicht	Ja, über Tools mit Freigabepflicht
Berechtigungsmodi	`manual`, `yolo`, `auto` (u. a. über `/permission` bzw. `default_permission_mode`)	Feiner gestuft: `default`, `acceptEdits`, `plan`, `auto`, `dontAsk`, `bypassPermissions` (über `/permissions` bzw. `permissions.defaultMode`)
Projektanweisungen	`AGENTS.md` (+ `.kimi-code/AGENTS.md`)	`CLAUDE.md` (+ `.claude/CLAUDE.md`, `CLAUDE.local.md`)
Skills	Ja, `SKILL.md` mit Frontmatter, `.kimi-code/skills/`	Ja, `SKILL.md` mit Frontmatter, `.claude/skills/`
Hooks	Ja, in `config.toml` (`[[hooks]]`, TOML)	Ja, in `settings.json` (`hooks`-Objekt, JSON)
Eigene Slash Commands	Kein direktes Äquivalent zu `.claude/commands/`; Skills übernehmen diese Rolle	`.claude/commands/` sowie Skills
Plugins	Ja, aktuell **nur benutzerweit** installierbar, kein Projekt-Scope	Ja, benutzer- und projektweit, mit Marketplace-Konzept
MCP	Ja, projekt- und benutzerweite `mcp.json`	Ja, projektweite `.mcp.json`, benutzerweite Konfiguration
Subagenten	Eingebaute Typen `coder`/`explore`/`plan`, zusätzlich Swarm-Modus (bis zu 128 gleichzeitige Subagenten) und persistenter Goal-Modus	Frei definierbare Subagenten unter `.claude/agents/`, automatische Delegation über Beschreibung
Sessions	`/sessions` (Alias `/resume`), `--continue`/`-c`, `--session`/`-S`	`/resume`, `--continue`/`-c`, `--resume`/`-r`, `--fork-session`
Automatisierung	`-p` für Einzel-Prompts, `kimi server`, Ausgabeformate `text`/`stream-json`	`-p`/`--print`, offizielle GitHub-Actions-Integration (`claude-code-action`), mehrere

Bereich	Kimi Code	Claude Code
		Ausgabeformate
Cloudmodelle	Kimi-Modelle sowie Anthropic-, OpenAI-kompatible, Google-Gemini- und Vertex-AI-Provider direkt konfigurierbar	Auf Claude-Modelle ausgerichtet
Lokale Modelle	Über den Provider-Typ `openai` direkt konfigurierbar	Nicht der primäre Einsatzzweck
Reife und Ökosystem	Jünger, kleineres Ökosystem, Plugin-System noch ohne Projekt-Scope	Länger etabliert, größeres Skill-/Plugin-Ökosystem, mehr offizielle Integrationen

Ein paar Unterschiede verdienen mehr Kontext als eine Tabellenzeile:

Der **Plan-Modus** funktioniert konzeptionell ähnlich (lesender Zugriff, kein Schreiben), ist bei Kimi Code aber ein expliziter Ein/Aus-Schalter (`/plan on`/`/plan off`), während Claude Code Plan als einen von mehreren Permission-Modi behandelt und dadurch mehr Zwischenstufen kennt (etwa `acceptEdits` für automatische Datei-Änderungen ohne volle Auto-Freigabe für Shell-Befehle).

Bei **Plugins** ist der Unterschied praktisch relevant: Claude-Code-Plugins können sowohl benutzer- als auch projektweit installiert werden, Kimi-Code-Plugins laut aktueller Dokumentation ausschließlich benutzerweit. Ein Kimi-Code-Plugin lässt sich also nicht einfach ins Projekt einchecken, damit es bei allen Teammitgliedern automatisch aktiv ist. Jede Person muss es einzeln installieren.

Bei **Subagenten** geht Kimi Code mit dem Swarm-Modus einen Schritt weiter als Claude Code: Bis zu 128 gleichzeitige Subagenten lassen sich auf eine Liste gleichartiger Teilaufgaben ansetzen. Das ist leistungsfähig, erhöht aber auch den Tokenverbrauch entsprechend (siehe Kapitel „Subagenten und autonome Arbeitsweisen“).

Claude Code ist enger mit den Claude-Modellen verbunden und gilt bei komplexen, langfristigen Entwicklungsaufgaben verbreitet als das ausgereiftere Werkzeug. Kimi Code ist bei der Wahl des Modellanbieters flexibler und kann OpenAI-kompatible lokale oder externe Endpunkte ohne Umwege direkt konfigurieren.

3 Wichtige Links

- Kimi Code Homepage
- Kimi Code Dokumentation – Einstieg
- Kimi Code auf GitHub
- Kimi Platform API
- Kimi Platform API Dokumentation
- Kimi-Mitgliedschaften und Preise
- LM Studio
- Ollama

> Die ältere, Python/uv-basierte Anwendung `kimi-cli` (Repository `github.com/MoonshotAI/kimi-cli`) wird laut offizieller Migrations-Dokumentation schrittweise durch die aktuelle, in Node.js geschriebene Kimi-Code-CLI (Repository `github.com/MoonshotAI/kimi-code`) ersetzt: „Kimi Code CLI has gone through a major version upgrade — moving from Python/uv to Node.js […] The legacy version will gradually be phased out.“ Neue Installationen sollten ausschließlich die aktuelle Dokumentation und das Verzeichnis `~/.kimi-code/` verwenden. Das alte Datenverzeichnis `~/.kimi/` gehört zu `kimi-cli` und wird von Kimi Code nur für eine optionale, einmalige Migration gelesen (siehe „Installation“).

4 Installation

macOS und Linux

Zwei offizielle Installationswege:

```
curl -fsSL https://code.kimi.com/kimi-code/install.sh | bash
```

Oder über Homebrew:

```
brew install kimi-code
```

Das Installationsskript lädt die aktuelle Version herunter und richtet den Befehl `kimi` ein. Eine vorhandene Node.js-Installation ist dafür nicht erforderlich.

Danach ein neues Terminal öffnen und die Installation prüfen:

```
kimi --version
```

Windows

Unter Windows wird zunächst Git for Windows benötigt, da Kimi Code die mitgelieferte Git-Bash als Shell verwendet. Installation in PowerShell:

```
irm https://code.kimi.com/kimi-code/install.ps1 | iex
```

Liegt Git Bash an einem nicht-standardmäßigen Ort, lässt sich der Pfad über die Umgebungsvariable `KIMI_SHELL_PATH` setzen.

Installation über npm oder pnpm

Vorausgesetzt wird Node.js 22.19.0 oder neuer:

```
npm install -g @moonshot-ai/kimi-code
```

Alternativ mit pnpm:

```
pnpm add -g @moonshot-ai/kimi-code
```

Aktualisieren

```
kimi upgrade
```

`kimi update` ist ein Alias für denselben Befehl. Bei einer npm-Installation lässt sich auch direkt aktualisieren:

```
npm install -g @moonshot-ai/kimi-code@latest
```

Deinstallation

Bei Installation über das Skript: die `kimi`-Executable manuell entfernen. Bei Installation über Homebrew: `brew uninstall kimi-code`. Bei npm-Installation:

```
npm uninstall -g @moonshot-ai/kimi-code
```

Abgrenzung zur alten CLI und Migration

Beim ersten Start prüft Kimi Code, ob unter dem alten Datenverzeichnis `~/.kimi/` (kimi-cli) Daten vorhanden sind, und bietet dann eine Migration an. Manuell lässt sich das jederzeit auslösen:

```
kimi migrate
```

Übernommen werden dabei unter anderem `config.toml`, MCP-Server-Konfiguration, Eingabe-Historie und Sessions. **Nicht** übernommen werden OAuth-Anmeldedaten und MCP-Autorisierungen (hier ist nach der Migration ein erneutes `/login` nötig) sowie kimi-cli-spezifische Plugins. Beide CLI-Versionen können parallel installiert bleiben; die alten Daten unter `~/.kimi/` werden dabei nicht verändert.

5 Erster Start und Anmeldung

Zuerst wird in den Projektordner gewechselt:

```
cd /pfad/zum/projekt
```

Für Änderungen sollte ein eigener Git-Branch verwendet werden:

```
git switch -c kimi-test
```

Danach wird Kimi Code gestartet:

```
kimi
```

Beim ersten Start öffnet folgender Command die Anmeldung:

```
/login
```

Dabei stehen zwei grundlegend unterschiedliche Wege zur Verfügung, die auch unterschiedlich abgerechnet werden:

- **Kimi Code (Mitgliedschaft):** Anmeldung per Kimi-Konto über einen OAuth-Flow im Browser. Der Verbrauch wird gegen das monatliche oder jährliche Kontingent des gebuchten Mitgliedschaftstarifs verrechnet (siehe Kapitel „Preise“). Diese Mitgliedschaft ist von der allgemeinen Kimi-Mitgliedschaft (Kimi-Chat in Web/App) zu unterscheiden. Moonshot plant, beide Kontingente künftig noch klarer in zwei getrennte Pläne aufzuteilen.
- **Kimi Platform:** Anmeldung mit einem API-Key der Kimi Platform. Der Verbrauch wird direkt nach Tokenmenge abgerechnet, unabhängig von einer Kimi-Code-Mitgliedschaft und deren Kontingent.

Weitere Cloudanbieter (Anthropic, OpenAI-kompatible Anbieter, Google Gemini, Vertex AI) sowie lokale Modellserver werden über folgenden Command eingerichtet:

```
/provider
```

Dabei gilt: Jeder zusätzliche Anbieter hat sein eigenes Abrechnungsmodell und sein eigenes Kontingent, unabhängig von einer eventuell vorhandenen Kimi-Code-Mitgliedschaft.

6 Ein Projekt einrichten

Im Projekt wird zunächst Kimi Code gestartet:

```
cd /pfad/zum/projekt
kimi
```

Danach kann Kimi Code die Projektstruktur analysieren und eine Anweisungsdatei erzeugen:

```
/init
```

Der Command erstellt eine `AGENTS.md`. Sie enthält Informationen über Architektur, Build-Befehle, Tests und Konventionen. Die erzeugte Datei sollte vor der weiteren Arbeit manuell geprüft werden.

Ein kompaktes Beispiel:

````
# Projektanweisungen

## Architektur

- Die Anwendung verwendet SwiftUI und Swift 6.4.
- Geschäftslogik befindet sich nicht direkt in Views.
- Neue Abhängigkeiten dürfen nur nach Rücksprache hinzugefügt werden.

## Build

```bash
xcodebuild \
  -project Example.xcodeproj \
  -scheme Example \
  -destination 'platform=iOS Simulator,name=iPhone 17 Pro' \
  build
```

## Tests

```bash
xcodebuild \
  -project Example.xcodeproj \
  -scheme Example \
  -destination 'platform=iOS Simulator,name=iPhone 17 Pro' \
  test
```

## Regeln

- Neue Funktionen benötigen Unit Tests.
- Öffentliche APIs dürfen nicht ohne Rücksprache verändert werden.
- Vor Abschluss einer Aufgabe müssen Build und Tests erfolgreich sein.
````

7 AGENTS.md und .kimi-code

Gibt es eine Entsprechung zu CLAUDE.md?

Ja. Die entsprechende Datei heißt bei Kimi Code:

```
AGENTS.md
```

Kimi Code unterstützt verschiedene Gültigkeitsbereiche:

Zweck	Datei
Projektweite Anweisungen	`AGENTS.md` im Projekt-Root
Kimi-spezifische Projektanweisungen	`.kimi-code/AGENTS.md`
Globale Kimi-Anweisungen	`~/.kimi-code/AGENTS.md` (genauer: `$KIMI_CODE_HOME/AGENTS.md`)
Werkzeugübergreifende globale Anweisungen	`~/.agents/AGENTS.md`

Die werkzeugübergreifende Datei `~/.agents/AGENTS.md` bleibt bewusst am tatsächlichen Betriebssystem-Home und wandert nicht mit einem eventuell umgeleiteten `KIMI_CODE_HOME` mit. Sie ist für Anweisungen gedacht, die mit mehreren Agent-Tools geteilt werden sollen, nicht nur mit Kimi Code.

Gibt es eine Entsprechung zu .claude?

Ja. Das Kimi-spezifische Projektverzeichnis heißt:

```
.kimi-code/
```

Eine mögliche Projektstruktur:

```
mein-projekt/
├── AGENTS.md
├── .kimi-code/
│   ├── AGENTS.md
│   ├── mcp.json
│   ├── local.toml
│   └── skills/
│       └── swift-review/
│           └── SKILL.md
├── Sources/
└── Tests/
```

Die wichtigsten Entsprechungen zu Claude Code:

Claude Code	Kimi Code
`CLAUDE.md`	`AGENTS.md`
`~/.claude/CLAUDE.md`	`~/.kimi-code/AGENTS.md`
`.claude/`	`.kimi-code/`

Claude Code	Kimi Code
`.claude/skills/`	`.kimi-code/skills/`
`.claude/commands/`	Skills oder Plugin-Commands
`.mcp.json`	`.kimi-code/mcp.json`
`settings.json`	`~/.kimi-code/config.toml`
Hooks in `settings.json`	`[[hooks]]` in `config.toml`

Welche Dateien eingecheckt werden sollten und welche lokal bleiben:

- **Einchecken:** `AGENTS.md`, `.kimi-code/AGENTS.md`, `.kimi-code/skills/` – das sind projektweite, für das ganze Team gedachte Anweisungen und Skills.
- **Mit Vorsicht einchecken:** `.kimi-code/mcp.json` – nur, wenn die Datei keine Zugangsdaten im Klartext enthält. Enthält sie Secrets, gehört sie nicht ins Repository.
- **Lokal halten:** `.kimi-code/local.toml` enthält Einstellungen für einen bestimmten Checkout und sollte nicht geteilt werden:

```
.kimi-code/local.toml
```

Globale Konfigurationen, Anmeldedaten, Sitzungen und Logs befinden sich standardmäßig unter:

```
~/.kimi-code/
```

8 Konfiguration

Kimi Code liest seine Einstellungen aus TOML-Dateien:

Datei	Ort	Zweck
`config.toml`	`~/.kimi-code/config.toml` (`$KIMI_CODE_HOME`)	Haupt-Konfiguration: Modelle, Provider, Berechtigungen, Hooks, Telemetrie
`tui.toml`	`~/.kimi-code/tui.toml`	Terminal-UI-Einstellungen (Theme, Editor, Benachrichtigungen, Auto-Update)
`local.toml`	`.kimi-code/local.toml` (Projekt)	Lokale, nicht geteilte Projekteinstellungen

config.toml

Bestätigte, tatsächlich verwendbare Felder (Auszug):

```
default_model = "kimi-code/k3"
default_permission_mode = "manual"   # manual | yolo | auto
default_plan_mode = false
telemetry = true                     # Standard true; nur explizites
false deaktiviert Telemetrie

[providers."managed:kimi-code"]
type = "kimi"
base_url = "https://api.kimi.com/coding/v1"
api_key = ""

[models."kimi-code/k3"]
provider = "managed:kimi-code"
model = "k3"
max_context_size = 1048576

[[permission.rules]]
decision = "allow"     # allow | deny | ask
pattern = "Read"

[[hooks]]
event = "PreToolUse"
matcher = "Bash"
command = "node ~/.kimi-code/hooks/check-bash.mjs"
timeout = 5
```

Provider-Einträge stehen unter `[providers.<name>]`, Modell-Einträge unter `[models.<name>]` und referenzieren einen Provider über das Feld `provider`. Enthält der Provider- oder Modellname einen Doppelpunkt oder Schrägstrich, muss der Tabellenname in Anführungszeichen stehen, wie im Beispiel `[providers."managed:kimi-code"]`.

Wichtig: Kimi Code liest API-Keys nicht automatisch aus der Shell-Umgebung. Ein einfaches `export OPENAI_API_KEY=...` wirkt sich ohne einen passenden Eintrag in `config.toml` nicht aus. Der Key muss entweder direkt im Provider-Eintrag stehen oder über die vom jeweiligen Provider-Typ vorgesehene Umgebungsvariable referenziert werden (siehe Kapitel „Cloud-LLMs verwenden“).

local.toml

`.kimi-code/local.toml` enthält projektspezifische, nicht geteilte Einstellungen. Offiziell dokumentiert ist aktuell vor allem das Feld für zusätzliche Arbeitsverzeichnisse, das automatisch über `/add-dir` befüllt wird:

```
[workspace]
additional_dir = ["../shared-lib"]
```

Die Datei sollte nicht eingecheckt werden (siehe Kapitel „AGENTS.md und .kimi-code“).

Telemetrie

Telemetrie lässt sich in `config.toml` deaktivieren:

```
telemetry = false
```

9 Im Projekt arbeiten

Projekt zunächst analysieren

Für ein unbekanntes oder größeres Projekt sollte zunächst der Plan-Modus verwendet werden:

```
/plan on
```

Beispiel:

```
Analysiere das Projekt. Prüfe Architektur, Build-Konfiguration,
Abhängigkeiten und Tests. Erstelle einen Plan für die Migration auf
Swift 6.4. Verändere noch keine Dateien.
```

Nach der Prüfung des Plans:

```
/plan off
```

Danach kann ein überschaubarer Arbeitsschritt umgesetzt werden:

```
Setze nur den ersten Abschnitt des Plans um. Ändere keine öffentlichen
Schnittstellen. Führe anschließend Build und Unit Tests aus und fasse
den Git-Diff zusammen.
```

Für länger laufende, selbstständige Arbeit an einem einzelnen Ziel eignet sich alternativ der Goal-Modus (`/goal`, siehe Kapitel „Subagenten und autonome Arbeitsweisen“). Dort steuert Kimi Code mehrere aufeinanderfolgende Turns selbst, statt nach jedem Schritt auf eine neue Anweisung zu warten.

Empfohlener Ablauf

1. Projekt analysieren lassen.
2. Plan erstellen und kontrollieren.
3. Änderungen in kleinen Schritten beauftragen.
4. Dateiänderungen und Shell-Befehle bewusst freigeben.
5. Build und Tests ausführen lassen.
6. `git diff` selbst prüfen.
7. Änderungen erst danach committen.

Der YOLO-Modus sollte in wichtigen Projekten deaktiviert bleiben:

```
/yolo off
```

10 Wichtige Commands

Eine vollständige Übersicht erscheint mit:

```
/help
```

Kimi Code kennt vier unterschiedliche Arten von Eingaben, die nicht verwechselt werden sollten:

- **Eingebaute Commands** – fest in der CLI codiert, zum Beispiel `/login`, `/model`, `/plan`.
- **Dynamische Skill-Commands** – automatisch aus Skills erzeugt (`/skill:name`, Kurzform `/name`), siehe Kapitel „Skills“.
- **Plugin-Commands** – von installierten Plugins registriert, in der Form `/<plugin-id>:<command>`.
- **Normale Prompts** – jeder Text ohne führenden Slash wird als Auftrag an das Modell interpretiert, nicht als Command.

Command	Alias	Funktion
`/login`	—	Bei Kimi Code oder der API anmelden
`/logout`	—	Aktuelle Anmeldung entfernen
`/init`	—	Projekt analysieren und `AGENTS.md` erzeugen
`/plan on` / `/plan off`	—	Plan-Modus aktivieren/beenden
`/model`	—	Modell wechseln
`/provider`	—	Modellanbieter verwalten
`/new`	`/clear`	Neue Sitzung beginnen
`/sessions`	`/resume`	Vorhandene Sitzung auswählen
`/compact`	—	Gesprächskontext zusammenfassen
`/usage`	—	Token- und Kontingentverbrauch anzeigen
`/permission`	—	Berechtigungsmodus einstellen
`/add-dir <pfad>`	—	Zusätzliches Arbeitsverzeichnis freigeben
`/mcp`	—	Status der MCP-Server anzeigen
`/mcp-config`	—	MCP-Server konfigurieren
`/plugins`	—	Plugins verwalten
`/yolo on` / `/yolo off`	`/yes`	Automatische Freigabe umschalten
`/exit`	`/quit`, `/q`	Kimi Code beenden

Die letzte Sitzung im aktuellen Projekt lässt sich fortsetzen:

```
kimi --continue
```

Eine einzelne nicht interaktive Aufgabe kann mit `-p` ausgeführt werden:

```
kimi -p "Führe die Unit Tests aus und fasse fehlgeschlagene Tests
zusammen."
```

Weitere häufig genutzte CLI-Flags:

Flag	Kurzform	Bedeutung
`--version`	`-V`	Version anzeigen
`--help`	`-h`	Hilfe anzeigen
`--continue`	`-c`	Letzte Session im aktuellen Verzeichnis fortsetzen
`--session <id>`	`-S`	Bestimmte Session fortsetzen
`--model <model>`	`-m`	Modell wählen
`--prompt <prompt>`	`-p`	Nicht-interaktiver Einzel-Prompt
`--yolo`	`-y`	Automatische Freigabe
`--plan`	—	Start im Plan-Modus

11 Skills

Skills enthalten wiederverwendbare Arbeitsabläufe, Regeln oder spezialisiertes Wissen.

Projektbezogene Skills:

```
.kimi-code/skills/
```

Globale Kimi-Skills:

```
~/.kimi-code/skills/
```

Werkzeugübergreifende Skills:

```
.agents/skills/
~/.agents/skills/
```

Beispielstruktur:

```
.kimi-code/
└── skills/
    └── swift-review/
        └── SKILL.md
```

Beispiel für `SKILL.md`:

```
---
name: swift-review
description: Prüft Swift- und SwiftUI-Code nach den Standards dieses
Projekts
type: prompt
whenToUse: Wenn Swift-Code erstellt, verändert oder überprüft wird
---

Prüfe den betroffenen Swift-Code insbesondere auf:

- Swift-6.4-Compilerprobleme
- Actor Isolation und Sendability
- Main-Actor-Verstöße
- unnötige Zustände in SwiftUI
- fehlende Fehlerbehandlung
- fehlende Unit Tests

Verändere den Code nur, wenn der Benutzer ausdrücklich eine Korrektur
verlangt.
```

Neben `name`, `description` und `whenToUse` unterstützt die Frontmatter weitere Felder: `type` (`prompt` als Standard; `flow` für Skills, die nur manuell aufrufbar sind und nicht automatisch vom Modell ausgewählt werden), `disableModelInvocation` (unterdrückt die automatische Aktivierung vollständig) und `arguments` (eine Liste benannter Parameter, die im Skill-Text über `$<name>` eingesetzt werden können). Neben der gezeigten Verzeichnisform (`<name>/SKILL.md`) ist auch eine flache Form (`<name>.md`) möglich; dort entfallen `name` und `description` und werden aus Dateiname beziehungsweise erster nicht leerer Zeile abgeleitet.

Manueller Aufruf:

```
/skill:swift-review
```

Mit zusätzlichem Auftrag:

```
/skill:swift-review Prüfe die Änderungen im aktuellen Git-Diff.
```

Wenn der Name nicht mit einem System-Command kollidiert, funktioniert auch die Kurzform:

```
/swift-review
```

Kimi Code kann einen Skill anhand von `description` und `whenToUse` automatisch auswählen, sofern `disableModelInvocation` nicht gesetzt ist und `type` nicht `flow` lautet.

Unterschiede zu Claude-Code-Skills

Claude Code verwendet ein sehr ähnliches Konzept: `SKILL.md`-Dateien mit Frontmatter unter `.claude/skills/`, automatische oder manuelle Aktivierung, Aufruf per `/skillname`. Der wesentliche Unterschied liegt in den unterstützten Frontmatter-Feldern (Claude Code kennt zusätzlich zum Beispiel `invocation-control`, `permissionMode` und `model` zur Modellwahl pro Skill) und darin, dass Kimi Code zusätzlich einen werkzeugübergreifenden, nicht toolspezifischen Speicherort (`.agents/skills/`, `~/.agents/skills/`) kennt.

12 Eigene Commands und Plugins

Kimi Code besitzt kein direktes, vollständig identisches Gegenstück zu `.claude/commands/`. Für einfache, projektbezogene Prompt-Commands sind Skills die einfachste Lösung. Sie werden automatisch als Slash Commands registriert (siehe Kapitel „Skills“).

Komplexere Command-Sammlungen lassen sich als Plugin verpacken, wahlweise ergänzt um MCP-Server, Hooks und Skills:

```
apple-development/
├── kimi.plugin.json
└── commands/
    └── migration-check.md
```

`kimi.plugin.json`:

```
{
  "name": "apple-development",
  "version": "1.0.0",
  "commands": "./commands/"
}
```

Das Manifest heißt `kimi.plugin.json` (alternativ `.kimi-plugin/plugin.json`) und unterstützt neben `name`, `version` und `commands` weitere Felder wie `description`, `keywords`, `author`, `homepage`, `license`, `skills` (eigene Skill-Verzeichnisse), `mcpServers` und `hooks`. Ein Plugin kann also mehrere Erweiterungsarten gleichzeitig mitbringen.

`commands/migration-check.md`:

```
---
description: Prüft ein Projekt auf notwendige Apple-27-Migrationen
---

Analysiere $ARGUMENTS und erstelle eine priorisierte
Migrationscheckliste.
Verändere keine Dateien.
```

Nach Installation und Aktivierung des Plugins:

```
/apple-development:migration-check Sources/
```

Plugins werden über `/plugins` verwaltet, unter anderem mit `/plugins install <pfad-oder-url>`, `/plugins list`, `/plugins enable|disable <id>`, `/plugins remove <id>` und `/plugins reload`.

Wichtige Einschränkung: Plugins werden aktuell ausschließlich benutzerweit installiert und gelten dann für alle Projekte dieses Benutzers. Eine projektbezogene Installation ist laut aktueller Dokumentation noch nicht vorgesehen. Ein Kimi-Code-Plugin lässt sich also, anders als ein Skill, nicht einfach ins Projekt einchecken, damit es bei allen Teammitgliedern automatisch aktiv ist. Jede Person muss es einzeln installieren.

13 Hooks

Hooks führen bei bestimmten Ereignissen lokale Skripte aus. Typische Einsatzbereiche sind:

- gefährliche Shell-Befehle erkennen
- nach Änderungen Formatter oder Prüfungen starten
- Desktop-Benachrichtigungen anzeigen
- zusätzliche Informationen in den Kontext aufnehmen
- Tool-Aufrufe protokollieren

Hooks werden in folgender Datei konfiguriert:

```
~/.kimi-code/config.toml
```

Benachrichtigung unter macOS

```
[[hooks]]
event = "Notification"
matcher = "task\\.completed"
command = "terminal-notifier -title Kimi -message 'Aufgabe
abgeschlossen'"
```

Dafür wird `terminal-notifier` benötigt:

```
brew install terminal-notifier
```

Prüfung vor Shell-Befehlen

```
[[hooks]]
event = "PreToolUse"
matcher = "Bash"
command = "node ~/.kimi-code/hooks/check-bash.mjs"
timeout = 5
```

Ein `[[hooks]]`-Eintrag akzeptiert genau vier Felder: `event` (Pflicht), `command` (Pflicht), `matcher` (optional, Regex, matcht alle Aufrufe wenn leer) und `timeout` (optional, 1–600 Sekunden, Standard 30). Zusätzliche Felder führen zu einem Konfigurationsfehler.

Wichtige Hook-Ereignisse (Auswahl; blockierend wirken davon nur `PreToolUse`, `UserPromptSubmit` und `Stop`):

- `UserPromptSubmit` (blockierend)
- `PreToolUse` (blockierend)
- `Stop` (blockierend)
- `PostToolUse`
- `PostToolUseFailure`
- `PermissionRequest`
- `PermissionResult`

- `SessionStart`
- `SessionEnd`
- `SubagentStart`
- `SubagentStop`
- `PreCompact` / `PostCompact`
- `Interrupt`
- `Notification`

Der Rückgabecode des Hook-Skripts entscheidet über das Ergebnis: `0` erlaubt die Aktion, `2` blockiert sie (die Begründung wird über stderr ausgegeben). Jeder andere Exit-Code sowie ein Timeout oder Absturz des Skripts führt dazu, dass die Aktion **trotzdem erlaubt wird**: Das ist das fail-open-Prinzip. Hooks ersetzen deshalb keine vollständige Sicherheitsbarriere. Wer sich allein auf einen Hook verlässt, um gefährliche Befehle zu verhindern, riskiert, dass ein defekter oder zu langsamer Hook genau diese Prüfung stillschweigend überspringt. Für wirklich harte Grenzen sind Berechtigungsregeln (`[[permission.rules]]`) und manuelle Kontrolle weiterhin notwendig.

14 MCP-Server

MCP-Server verbinden Kimi Code mit externen Werkzeugen, Datenquellen oder Unternehmensdiensten.

Interaktive Einrichtung:

```
/mcp-config
```

Projektbezogene MCP-Konfiguration:

```
.kimi-code/mcp.json
```

Globale MCP-Konfiguration:

```
~/.kimi-code/mcp.json
```

Status prüfen:

```
/mcp
```

Kimi Code unterstützt drei Verbindungsarten: `stdio` (lokaler Prozess, Feld `command`), `HTTP` (Feld `url`) und `SSE` (Feld `url` plus `transport: "sse"`). Für Server mit eigener Anmeldung steht ein OAuth-Login-Flow zur Verfügung:

```
/mcp-config login <server-name>
```

MCP-Zugangsdaten dürfen nicht unverschlüsselt in ein öffentliches Repository eingecheckt werden.

Im Vergleich zu Claude Code (`.mcp.json` im Projekt, Verwaltung über `claude mcp add` und `/mcp`) ist das Konzept sehr ähnlich: eine projektweite und eine benutzerweite Konfigurationsdatei, ein Status-Command und eine interaktive Einrichtung. Der Unterschied liegt vor allem im Dateinamen (`mcp.json` statt `.mcp.json`) und im Speicherort der benutzerweiten Variante (`~/.kimi-code/mcp.json` als eigene Datei statt eingebettet in die Claude-Code-Benutzer-Konfiguration).

15 Subagenten und autonome Arbeitsweisen

Kimi Code kann Teilaufgaben an Subagenten delegieren. Drei eingebaute Typen sind dokumentiert:

- `coder` – Standardtyp, mit vollem Werkzeugzugriff (Dateien, Shell)
- `explore` – nur lesender Zugriff, für Recherche und Codebase-Analyse
- `plan` – reine Planungsaufgaben, ohne Shell-Zugriff

Der Haupt-Agent entscheidet automatisch, wann eine Aufgabe an einen Subagenten delegiert wird. Jeder Dispatch erscheint standardmäßig als eigene Freigabeanfrage, sofern nicht eine passende Allow-Regel oder der YOLO-Modus ihn automatisch zulässt. Subagenten laufen in einem eigenen, vollständig isolierten Kontext und erben die Berechtigungen des Haupt-Agenten.

Swarm-Modus

Über `/swarm on|off` beziehungsweise `/swarm <aufgabe>` lässt sich eine größere Zahl gleichzeitiger Subagenten starten, die dieselbe Aufgabenvorlage auf eine Liste von Elementen anwenden, bis zu 128 gleichzeitige Subagenten. Die Obergrenze der Parallelität lässt sich über die Umgebungsvariable `KIMI_CODE_AGENT_SWARM_MAX_CONCURRENCY` begrenzen.

Goal-Modus

`/goal` ist ein eigenständiges Feature, kein Teil des Swarm-Modus: Statt mehrerer paralleler Agenten verfolgt Kimi Code hier ein einzelnes, persistentes Ziel über mehrere automatisch fortgesetzte Turns hinweg selbstständig, bis das Ziel erreicht oder abgebrochen wird. Verwaltung über `/goal status`, `/goal pause`, `/goal resume` und `/goal cancel`.

Risiken und Einsatzbereiche

Automatische Delegation und insbesondere der Swarm-Modus können den Tokenverbrauch deutlich erhöhen, da jeder Subagent einen eigenen Kontext aufbaut und eigene Modellaufrufe durchführt. Für gut abgegrenzte, wiederholbare Aufgaben, etwa dieselbe Prüfung über viele Dateien hinweg, ist das sinnvoll. Für offene, wenig spezifizierte Aufgaben steigt dagegen das Risiko unnötig hoher Kosten und schwer nachvollziehbarer Änderungen. In produktiven oder sicherheitsrelevanten Projekten sollte der YOLO-Modus auch bei aktiven Subagenten deaktiviert bleiben, damit Freigabeanfragen für Datei- und Shell-Zugriffe erhalten bleiben.

Claude Code kennt ein vergleichbares, aber anders aufgebautes Konzept: frei konfigurierbare Subagenten unter `.claude/agents/`, ebenfalls mit automatischer Delegation anhand einer Beschreibung und eigenem Kontextfenster. Einen dedizierten Swarm-Modus mit fester Obergrenze von 128 gleichzeitigen Subagenten wie bei Kimi Code gibt es in der Claude-Code-Dokumentation in dieser Form nicht.

16 Cloud-LLMs verwenden

Bei einem Cloud-LLM werden Prompts, Projektkontext und die für eine Aufgabe benötigten Dateiinhalte an einen externen Modellanbieter übertragen.

Kimi-Code-Mitgliedschaft

Kimi Code im Projekt starten:

```
cd /pfad/zum/projekt
kimi
```

Anmeldung öffnen:

```
/login
```

Danach `Kimi Code` auswählen und die Anmeldung im Browser abschließen.

Das gewünschte Modell wird über folgenden Command ausgewählt:

```
/model
```

Über `/model` lassen sich die für den gebuchten Tarif freigeschalteten Modelle interaktiv auswählen, unter anderem K3, Kimi K2.7 Code und Kimi K2.7 Code HighSpeed. Welche Modelle sichtbar sind, hängt vom Mitgliedschaftstarif ab: Die HighSpeed-Variante ist laut offiziellem Kimi-Code-Änderungsprotokoll beispielsweise erst ab dem Tarif Allegretto enthalten (siehe Kapitel „Preise“).

Kontingent und Tokenverbrauch:

```
/usage
```

Kimi Platform API

Für eine verbrauchsabhängige Abrechnung wird ein API-Key der Kimi Platform benötigt. Die Einrichtung kann über `/login` oder `/provider` erfolgen.

Die offiziellen Modell-IDs für Coding-Aufgaben lauten `kimi-k3`, `kimi-k2.7-code` und `kimi-k2.7-code-highspeed` sowie, als weiterhin angebotene, ältere Variante, `kimi-k2.6`. Kontextfenster: K3 1.048.576 Token (1M), K2.7 Code und K2.7 Code HighSpeed je 262.144 Token (256K).

Manuelle Konfiguration in `~/.kimi-code/config.toml`:

```
default_model = "kimi-cloud"

[providers.moonshot]
type = "kimi"
base_url = "https://api.moonshot.ai/v1"
api_key = "DEIN_API_KEY"

[models.kimi-cloud]
provider = "moonshot"
model = "kimi-k2.7-code"
```

```
max_context_size = 262144
```

API-Keys dürfen nicht in ein Git-Repository übernommen werden.

Andere Cloudanbieter

Kimi Code unterstützt außerdem folgende Provider-Typen (Feld `type` in `config.toml`): `anthropic` (Anthropic/Claude), `openai` und `openai_responses` (OpenAI sowie beliebige OpenAI-kompatible Anbieter, einschließlich lokaler Server), `google-genai` (Google Gemini) und `vertexai` (Google Vertex AI, über den Google-ADC-Anmeldeflow mit `GOOGLE_CLOUD_PROJECT`/`GOOGLE_CLOUD_LOCATION`).

Beispiel für einen OpenAI-kompatiblen Anbieter:

```
[providers.cloud-provider]
type = "openai"
base_url = "https://api.example.com/v1"
api_key = "DEIN_API_KEY"

[models.cloud-model]
provider = "cloud-provider"
model = "MODELLNAME"
max_context_size = 131072
```

Aktuelle Modellnamen und Kontextgrößen müssen aus der Dokumentation des jeweiligen Anbieters übernommen werden.

Zu beachten: Kimi Code liest API-Keys nicht automatisch aus exportierten Shell-Umgebungsvariablen. Ein `export ANTHROPIC_API_KEY=...` ohne passenden Eintrag in `config.toml` bleibt wirkungslos. Der Key muss im Provider-Eintrag stehen oder über die vom jeweiligen Provider-Typ vorgesehene Umgebungsvariable eingebunden werden (zum Beispiel `ANTHROPIC_API_KEY`/`ANTHROPIC_BASE_URL` für `anthropic`, `OPENAI_API_KEY`/`OPENAI_BASE_URL` für `openai`, `KIMI_API_KEY`/`KIMI_BASE_URL` für `kimi`).

Vor- und Nachteile

Vorteile:

- hohe Modellqualität
- große Kontextfenster
- meist zuverlässiges Tool Calling
- keine eigene GPU-Infrastruktur
- schnelle Einrichtung

Nachteile:

- Übertragung von Projektinformationen an einen Anbieter
- laufende Abonnement- oder Tokenkosten
- Internetverbindung erforderlich
- Rate Limits und Kontingente

- zusätzliche Datenschutz- und Compliance-Anforderungen

17 Lokale LLMs verwenden

Bei einem lokalen LLM läuft das Modell auf dem Entwicklungsrechner oder auf einem internen Server. Kimi Code kommuniziert über eine lokale OpenAI-kompatible API mit dem Modell. Kimi Code selbst bleibt dabei zwar eine lokal installierte CLI, aber „lokal“ im Sinn von „das Modell läuft komplett offline auf diesem Rechner“ trifft nur zu, wenn tatsächlich ein lokaler Modellserver konfiguriert ist. Bei einem Cloud-Modell ist die Bezeichnung „lokales Programm“ für Kimi Code irreführend, da die eigentliche Verarbeitung weiterhin extern stattfindet.

Kimi Code enthält selbst keine Modell-Laufzeit. Dafür kann beispielsweise folgende Software verwendet werden:

- LM Studio
- Ollama
- llama.cpp
- vLLM
- ein interner OpenAI-kompatibler Modellserver

Lokales Modell mit LM Studio

1. LM Studio installieren

LM Studio herunterladen und installieren.

2. Modell herunterladen

In LM Studio ein geeignetes Coding-Modell auswählen. Es sollte Programmierung, ein ausreichendes Kontextfenster und möglichst zuverlässiges Tool Calling unterstützen. Laut offizieller Dokumentation funktioniert natives Tool-Calling-Parsing aktuell besonders zuverlässig bei Modellen wie Qwen 2.5, Llama 3.1/3.2 und Ministral 8B; bei anderen Modellen läuft Tool Calling über einen generischeren Prompt-Mechanismus, der bei kleineren oder nicht dafür trainierten Modellen fehlerhaft formatierte Tool-Aufrufe liefern kann.

3. Modell laden

Das Modell laden und eine Quantisierung verwenden, die zum verfügbaren Arbeitsspeicher passt.

4. Lokalen Server starten

In LM Studio den lokalen API-Server starten (Developer-Tab, Schalter „Start Server“), alternativ über die mitgelieferte CLI:

```
lms server start
```

Der Standardendpunkt lautet:

```
http://localhost:1234/v1
```

Die angezeigte Modell-ID wird für Kimi Code benötigt und ist über `GET /v1/models` oder in der LM-Studio-Oberfläche abrufbar.

5. Kimi Code konfigurieren

`~/.kimi-code/config.toml` ergänzen:

```
default_model = "local-coder"

[providers.lm-studio]
type = "openai"
base_url = "http://localhost:1234/v1"
api_key = "local"

[models.local-coder]
provider = "lm-studio"
model = "MODELL-ID-AUS-LM-STUDIO"
max_context_size = 32768
```

`MODELL-ID-AUS-LM-STUDIO` muss durch die tatsächlich angezeigte Modell-ID ersetzt werden. LM Studio verlangt standardmäßig **keine** Authentifizierung, daher genügt für `api_key` ein beliebiger nicht leerer Platzhalterwert; eine offiziell vorgeschriebene Zeichenkette dafür gibt es nicht. Wird in LM Studio eine Authentifizierung aktiviert (Developer-Seite → Server Settings → Manage Tokens), muss stattdessen das dort erzeugte Token eingetragen werden.

6. Verbindung testen

```
cd /pfad/zum/projekt
kimi
```

Danach das lokale Modell auswählen:

```
/model
```

Ein sicherer erster Test:

```
Analysiere die Projektstruktur. Verändere keine Dateien und führe
keine Shell-Befehle aus. Nenne die wichtigsten Module des Projekts.
```

Schreibzugriffe sollten erst freigegeben werden, wenn Lesen, Kontextverarbeitung und Tool Calling zuverlässig funktionieren.

Lokales Modell mit Ollama

Ollama lässt sich über das offizielle Installationsskript einrichten (Linux):

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

Unter macOS und Windows steht ein Installer zum Download bereit. Danach ein Modell laden, zum Beispiel eine kleine, für einen gewöhnlichen Laptop praktikable Coding-Variante:

```
ollama pull qwen2.5-coder:0.5b
```

Ollama bietet ebenfalls eine OpenAI-kompatible API:

```
default_model = "ollama-coder"

[providers.ollama]
type = "openai"
```

```
base_url = "http://127.0.0.1:11434/v1"
api_key = "ollama"

[models.ollama-coder]
provider = "ollama"
model = "qwen2.5-coder:0.5b"
max_context_size = 65536
```

Der Wert `"ollama"` für `api_key` wird von Ollama zwar inhaltlich ignoriert, ist aber der offiziell dokumentierte Beispielwert für OpenAI-kompatible Clients, die zwingend einen Key-Parameter erwarten.

Ollama vergibt automatisch ein Kontextlimit passend zum verfügbaren VRAM: unter 24 GB VRAM 4.000 Token, bei 24–48 GB 32.000 Token, ab 48 GB 256.000 Token. Für agentische Coding-Workflows empfiehlt die Dokumentation mindestens 64.000 Token; das Limit lässt sich manuell überschreiben, zum Beispiel:

```
OLLAMA_CONTEXT_LENGTH=65536 ollama serve
```

Tool Calling wird über denselben `tools`-Parameter unterstützt (nicht jedoch `tool_choice`, `logit_bias` oder `n`). Die Ollama-Dokumentation empfiehlt Single-Shot-Tool-Calling ausdrücklich nur für Modelle, die zuverlässig genau einen Tool-Aufruf pro Antwort liefern (Beispiel `qwen3`); bei anderen Modellen ist mit geringerer Zuverlässigkeit zu rechnen.

Praktisches Beispiel: Ollama und Kimi Code automatisiert betreiben

Die manuellen Einzelschritte – Ollama installieren, Server starten, Modell laden, `config.toml` von Hand ergänzen, `kimi` mit dem richtigen Modellnamen aufrufen – wiederholen sich bei jedem Neustart. Für den lokalen Alltagsbetrieb lohnt sich deshalb eine kleine Automatisierung. Das folgende Beispiel besteht aus fünf Bash-Scripten plus einer gemeinsamen Hilfsbibliothek in einem Ordner `scripts/`; alle Aufrufe erfolgen aus diesem Ordner heraus. Voraussetzung ist macOS mit installiertem Homebrew (wird nur gebraucht, falls Ollama oder Kimi Code noch fehlen).

Script	Zweck
`ollama-start.sh`	Installiert Ollama bei Bedarf, startet den Server (falls er nicht schon läuft) und lädt das Standardmodell nach.
`ollama-select.sh`	Listet installierte Modelle auf, lässt eines interaktiv auswählen und startet es (`ollama run`); startet den Server bei Bedarf mit.
`kimi-code-local.sh`	Stellt Ollama sicher, trägt Provider/Modell automatisch in `~/.kimi-code/config.toml` ein und startet `kimi --model ollama-coder`.
`ollama-stop.sh`	Stoppt einen von `ollama-start.sh` gestarteten Server; nur mit `--force` auch einen fremd gestarteten.
`ollama-force-stop.sh`	Kurzform für `ollama-stop.sh --force` – beendet jeden laufenden `ollama serve`-Prozess.

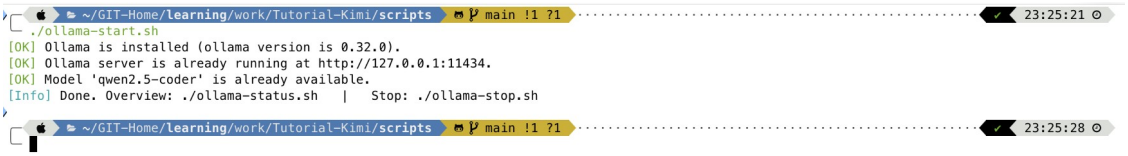
Script	Zweck
`lib/common.sh`	Gemeinsame Konfiguration (Modellname, Port, Kontextlänge, Pfade) und Hilfsfunktionen; wird von den anderen Scripten per `source` eingebunden, nicht selbst aufgerufen.

Alle Werte lassen sich per Umgebungsvariable überschreiben, unter anderem `OLLAMA_MODEL` (Standard `qwen2.5-coder`), `OLLAMA_CONTEXT_LENGTH` (Standard `65536`), `OLLAMA_PORT` (Standard `11434`) und `KIMI_CODE_HOME` (Standard `~/.kimi-code`).

1. Server starten

```
./ollama-start.sh
```

Das Script erkennt, ob Ollama bereits installiert ist, installiert es sonst per Homebrew, startet den Server (sofern er nicht schon über die Ollama.app oder Menüleiste läuft) und lädt das konfigurierte Modell nach, falls es lokal noch fehlt.

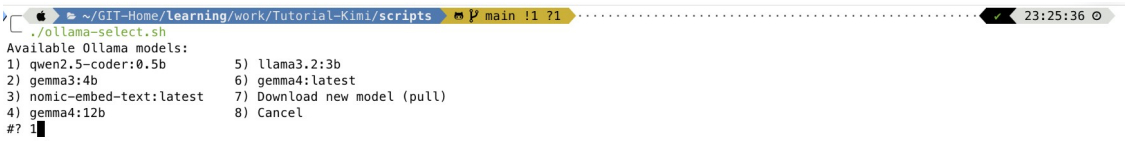


Ollama-Server wird über ollama-start.sh gestartet

2. Modell auswählen oder herunterladen

```
./ollama-select.sh
```

Zeigt alle lokal installierten Modelle in einer nummerierten Liste an, lässt eines auswählen und startet es interaktiv mit `ollama run`. Über den Menüpunkt „Download new model (pull)“ lässt sich zusätzlich ein neues Modell nachladen, ohne die bestehende Kimi-Code-Konfiguration zu verändern.



Modellauswahl über ollama-select.sh

3. Kimi Code mit dem lokalen Modell starten

```
./kimi-code-local.sh
```

Das Script stellt zunächst sicher, dass der Ollama-Server läuft (ruft dafür intern `ollama-start.sh` auf), installiert bei Bedarf die Kimi-Code-CLI per Homebrew und ergänzt anschließend – idempotent und mit automatischem Backup einer vorhandenen Datei als `config.toml.bak.<Zeitstempel>` – folgenden Block in `~/.kimi-code/config.toml`:

```
default_model = "ollama-coder"

# --- Local Ollama model (automatically inserted by kimi-code-local.sh)
---
[providers.ollama]
```

```
type = "openai"
base_url = "http://127.0.0.1:11434/v1"
api_key = "ollama"

[models.ollama-coder]
provider = "ollama"
model = "qwen2.5-coder:0.5b"
max_context_size = 65536

[thinking]
enabled = false
```

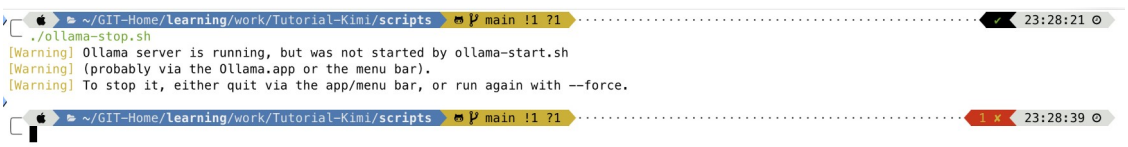
`default_model` und der `[thinking]`-Abschnitt stammen dabei aus der übrigen, bereits vorhandenen Kimi-Code-Konfiguration (z. B. aus einem vorherigen `kimi`-Lauf) – das Script selbst ergänzt ausschließlich den markierten `[providers.ollama]`/`[models.ollama-coder]`-Block. Anschließend startet das Script `kimi --model ollama-coder`; alle zusätzlichen Argumente werden 1:1 an `kimi` durchgereicht, z. B. `./kimi-code-local.sh -p "Kurze Aufgabe"`.

Erste Frage an Kimi Code mit lokalem Modell

4. Server wieder stoppen

```
./ollama-stop.sh
```

Beendet ausschließlich einen Server, der zuvor von `ollama-start.sh` selbst gestartet wurde – ein fremd gestarteter Server (z. B. über die Ollama.app) bleibt unangetastet, um kein versehentliches Beenden fremder Prozesse auszulösen. Mit `./ollama-stop.sh --force` oder gleichbedeutend `./ollama-force-stop.sh` lässt sich jeder laufende `ollama serve`-Prozess beenden, unabhängig davon, wie er gestartet wurde.



Ollama-Server wird gestoppt

Zusammengefasster Schnellstart:

```
./ollama-start.sh
./kimi-code-local.sh
```

Scriptquellen

Zur vollständigen Nachvollziehbarkeit der vollständige Quellcode aller sechs Dateien.

`scripts/lib/common.sh`:

```
#!/usr/bin/env bash
# Shared configuration and helper functions for the Ollama/Kimi Code
scripts.
# Sourced by the other scripts via `source`; not meant to be run
standalone.

set -euo pipefail

SCRIPT_LIB_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
SCRIPTS_DIR="$(cd "$SCRIPT_LIB_DIR/.." && pwd)"
RUN_DIR="$SCRIPTS_DIR/.run"
mkdir -p "$RUN_DIR"

OLLAMA_PORT="${OLLAMA_PORT:-11434}"
OLLAMA_URL="http://127.0.0.1:${OLLAMA_PORT}"
OLLAMA_MODEL="${OLLAMA_MODEL:-qwen2.5-coder}"
OLLAMA_CONTEXT_LENGTH="${OLLAMA_CONTEXT_LENGTH:-65536}"
OLLAMA_PID_FILE="$RUN_DIR/ollama.pid"
OLLAMA_LOG_FILE="$RUN_DIR/ollama.log"
OLLAMA_STATUS_HTML="$RUN_DIR/status.html"

KIMI_CODE_HOME="${KIMI_CODE_HOME:-$HOME/.kimi-code}"
KIMI_CODE_CONFIG="$KIMI_CODE_HOME/config.toml"
KIMI_OLLAMA_PROVIDER_NAME="ollama"
KIMI_OLLAMA_MODEL_NAME="ollama-coder"
KIMI_MARKER="# --- Local Ollama model (automatically inserted by kimi-
code-local.sh) ---"

# Colored output only if the terminal supports it
if [ -t 1 ]; then
    C_INFO=$'\033[36m'; C_OK=$'\033[32m'; C_WARN=$'\033[33m';
C_ERR=$'\033[31m'; C_RESET=$'\033[0m'
else
    C_INFO=""; C_OK=""; C_WARN=""; C_ERR=""; C_RESET=""
fi

info()  { printf '%s[Info]%s %s\n'  "$C_INFO" "$C_RESET" "$1"; }
ok()    { printf '%s[OK]%s %s\n'    "$C_OK"   "$C_RESET" "$1"; }
warn()  { printf '%s[Warning]%s %s\n' "$C_WARN" "$C_RESET" "$1"; }
err()   { printf '%s[Error]%s %s\n' "$C_ERR"  "$C_RESET" "$1" >&2; }

# Checks whether the Ollama server responds on OLLAMA_URL
is_ollama_running() {
    curl -fsS --max-time 2 "$OLLAMA_URL" >/dev/null 2>&1
}

# Waits up to $1 seconds for the Ollama server to respond
```

```
wait_for_ollama() {
    local timeout="${1:-30}"
    local waited=0
    while ! is_ollama_running; do
        if [ "$waited" -ge "$timeout" ]; then
            return 1
        fi
        sleep 1
        waited=$((waited + 1))
    done
    return 0
}

# Checks whether a given model is already available locally (with or
without tag)
ollama_has_model() {
    local model="$1"
    ollama list 2>/dev/null | awk 'NR>1 {print $1}' | grep -qE "^$
{model}(:[^[:space:]]+)?$"
}
```

scripts/ollama-start.sh:

```
#!/usr/bin/env bash
# Installs Ollama if needed, starts the server (if it isn't already
running)
# and pulls the configured default model if it's still missing.
#
# Usage:
#   ./ollama-start.sh                    # default model (qwen2.5-
coder)
#   OLLAMA_MODEL=llama3.1 ./ollama-start.sh
#   OLLAMA_CONTEXT_LENGTH=131072 ./ollama-start.sh

set -euo pipefail
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
# shellcheck source=lib/common.sh
source "$SCRIPT_DIR/lib/common.sh"

# 1. Check installation, install via Homebrew if needed
if ! command -v ollama >/dev/null 2>&1; then
    warn "Ollama is not installed."
    if command -v brew >/dev/null 2>&1; then
        info "Installing Ollama via Homebrew ..."
        brew install ollama
    else
        err "Homebrew was not found. Please install Ollama manually:"
        err "  https://ollama.com/download"
        exit 1
    fi
else
    ok "Ollama is installed ($(ollama --version 2>/dev/null | head
-n1))."
fi

# 2. Start the server if it isn't already running
if is_ollama_running; then
```

```
    ok "Ollama server is already running at $OLLAMA_URL."
else
    info "Starting Ollama server (context length: $
{OLLAMA_CONTEXT_LENGTH} tokens) ..."
    OLLAMA_CONTEXT_LENGTH="$OLLAMA_CONTEXT_LENGTH" \
        nohup ollama serve >"$OLLAMA_LOG_FILE" 2>&1 &
    echo $! >"$OLLAMA_PID_FILE"
    disown

    if wait_for_ollama 30; then
        ok "Ollama server reachable at $OLLAMA_URL (PID $(cat
"$OLLAMA_PID_FILE"), log: $OLLAMA_LOG_FILE)."
    else
        err "Ollama server is not responding after 30 seconds. Check
log: $OLLAMA_LOG_FILE"
        exit 1
    fi
fi

# 3. Pull the default model if it's still missing
if ollama_has_model "$OLLAMA_MODEL"; then
    ok "Model '$OLLAMA_MODEL' is already available."
else
    info "Downloading model '$OLLAMA_MODEL' (may take a few minutes
depending on size) ..."
    ollama pull "$OLLAMA_MODEL"
    ok "Model '$OLLAMA_MODEL' has been downloaded."
fi

info "Done. Overview: ./ollama-status.sh   |   Stop: ./ollama-stop.sh"
```

scripts/ollama-select.sh:

```
#!/usr/bin/env bash
# Lists locally installed Ollama models, lets the user pick one
interactively
# and starts it (ollama run). Starts the server automatically if needed.
#
# Usage:
#   ./ollama-select.sh

set -euo pipefail
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
# shellcheck source=lib/common.sh
source "$SCRIPT_DIR/lib/common.sh"

if ! command -v ollama >/dev/null 2>&1; then
    err "Ollama is not installed. Run first: ./ollama-start.sh"
    exit 1
fi

if ! is_ollama_running; then
    warn "Ollama server is not running. Starting it via ./ollama-
start.sh ..."
    "$SCRIPT_DIR/ollama-start.sh"
fi
```

```
MODELS=()
while IFS= read -r line; do
    MODELS+=("$line")
done < <(ollama list 2>/dev/null | awk 'NR>1 {print $1}')

if [ "${#MODELS[@]}" -eq 0 ]; then
    warn "No local models found."
    read -rp "Enter model name to download (e.g. qwen2.5-coder): "
NEW_MODEL
    ollama pull "$NEW_MODEL"
    exec ollama run "$NEW_MODEL"
fi

echo "Available Ollama models:"
select MODEL in "${MODELS[@]}" "Download new model (pull)" "Cancel"; do
    case "$MODEL" in
        "Cancel")
            info "Cancelled."
            exit 0
            ;;
        "Download new model (pull)")
            read -rp "Model name (e.g. llama3.1, mistral, phi3): "
NEW_MODEL
            ollama pull "$NEW_MODEL"
            exec ollama run "$NEW_MODEL"
            ;;
        "")
            warn "Invalid selection, please try again."
            ;;
        *)
            ok "Loading model: $MODEL"
            exec ollama run "$MODEL"
            ;;
    esac
done
```

scripts/kimi-code-local.sh:

```
#!/usr/bin/env bash
# Sets up Kimi Code for a purely local Ollama model and starts the CLI
with it.
# Automated: ensure the Ollama server is up, pull the model, register
the
# provider/model in ~/.kimi-code/config.toml (idempotent, with backup if
the
# file already exists), then start `kimi --model ollama-coder`.
#
# Usage:
#   ./kimi-code-local.sh                       # starts Kimi Code
interactively, local
#   ./kimi-code-local.sh -p "Short task"        # all arguments are
passed through to `kimi`
#   OLLAMA_MODEL=llama3.1 ./kimi-code-local.sh  # use a different local
model

set -euo pipefail
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
```

```
# shellcheck source=lib/common.sh
source "$SCRIPT_DIR/lib/common.sh"

# 1. Ensure the Ollama server is up (install/start/pull model is handled
by ollama-start.sh)
info "Checking/starting Ollama ..."
"$SCRIPT_DIR/ollama-start.sh"

# 2. Ensure the Kimi Code CLI is installed
if ! command -v kimi >/dev/null 2>&1; then
    warn "Kimi Code (command 'kimi') is not installed."
    if command -v brew >/dev/null 2>&1; then
        info "Installing Kimi Code via Homebrew ..."
        brew install kimi-code
    else
        err "Homebrew was not found. Please install Kimi Code manually:"
        err "  curl -fsSL https://code.kimi.com/kimi-code/install.sh |
bash"
        exit 1
    fi
else
    ok "Kimi Code is installed ($(kimi --version 2>/dev/null | head
-n1))."
fi

# 3. Register provider/model in config.toml (idempotent)
mkdir -p "$KIMI_CODE_HOME"

if [ -f "$KIMI_CODE_CONFIG" ] && grep -qF "$KIMI_MARKER"
"$KIMI_CODE_CONFIG"; then
    ok "Local Ollama model is already registered in $KIMI_CODE_CONFIG."
else
    if [ -f "$KIMI_CODE_CONFIG" ]; then
        BACKUP="${KIMI_CODE_CONFIG}.bak.$(date '+%Y%m%d%H%M%S')"
        cp "$KIMI_CODE_CONFIG" "$BACKUP"
        info "Existing config.toml backed up to $BACKUP."
    fi
    {
        echo ""
        echo "$KIMI_MARKER"
        echo "[providers.${KIMI_OLLAMA_PROVIDER_NAME}]"
        echo "type = \"openai\""
        echo "base_url = \"${OLLAMA_URL}/v1\""
        echo "api_key = \"ollama\""
        echo ""
        echo "[models.${KIMI_OLLAMA_MODEL_NAME}]"
        echo "provider = \"${KIMI_OLLAMA_PROVIDER_NAME}\""
        echo "model = \"${OLLAMA_MODEL}\""
        echo "max_context_size = ${OLLAMA_CONTEXT_LENGTH}"
    } >>"$KIMI_CODE_CONFIG"
    ok "Provider '${KIMI_OLLAMA_PROVIDER_NAME}' and model '$
{KIMI_OLLAMA_MODEL_NAME}' registered in $KIMI_CODE_CONFIG."
fi

# 4. Start Kimi Code with the local model, passing through all arguments
info "Starting 'kimi --model ${KIMI_OLLAMA_MODEL_NAME}' ..."
```

```
exec kimi --model "$KIMI_OLLAMA_MODEL_NAME" "$@"
```

scripts/ollama-stop.sh:

```
#!/usr/bin/env bash
# Stops an Ollama server that was started by ollama-start.sh.
#
# Usage:
#   ./ollama-stop.sh          # only stops what ollama-start.sh itself
started
#   ./ollama-stop.sh --force  # also terminates any running "ollama
serve" process
#                             # (e.g. one started via the
Ollama.app/menu bar)

set -euo pipefail
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
# shellcheck source=lib/common.sh
source "$SCRIPT_DIR/lib/common.sh"

FORCE=0
[ "${1:-}" = "--force" ] && FORCE=1

if ! is_ollama_running; then
    ok "Ollama server is not running (unreachable at $OLLAMA_URL)."
    rm -f "$OLLAMA_PID_FILE"
    exit 0
fi

if [ -f "$OLLAMA_PID_FILE" ] && kill -0 "$(cat "$OLLAMA_PID_FILE")"
2>/dev/null; then
    PID="$(cat "$OLLAMA_PID_FILE")"
    info "Stopping Ollama server started by this script (PID $PID) ..."
    kill "$PID" 2>/dev/null || true
    for _ in $(seq 1 10); do
        kill -0 "$PID" 2>/dev/null || break
        sleep 1
    done
    if kill -0 "$PID" 2>/dev/null; then
        warn "Process is not responding to SIGTERM, sending SIGKILL."
        kill -9 "$PID" 2>/dev/null || true
    fi
    rm -f "$OLLAMA_PID_FILE"
    ok "Ollama server stopped."
elif [ "$FORCE" -eq 1 ]; then
    warn "No process started by this script is known, terminating anyway
via --force."
    pkill -f "ollama serve" 2>/dev/null || true
    sleep 1
    if is_ollama_running; then
        err "Ollama server is still running (possibly restarted via the
Ollama.app)."
        exit 1
    fi
    ok "Ollama server stopped."
else
```

```
    warn "Ollama server is running, but was not started by ollama-
start.sh"
    warn "(probably via the Ollama.app or the menu bar)."
    warn "To stop it, either quit via the app/menu bar, or run again
with --force."
    exit 1
fi
```

scripts/ollama-force-stop.sh:

```
#!/usr/bin/env bash
# Terminates any running "ollama serve" process, whether started by
# ollama-start.sh or via the Ollama.app/menu bar.
#
# Shorthand for: ./ollama-stop.sh --force
#
# Usage:
#   ./ollama-force-stop.sh

set -euo pipefail
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

exec "$SCRIPT_DIR/ollama-stop.sh" --force
```

Kimi-Modelle lokal betreiben

Einige Kimi-Modellgewichte sind grundsätzlich selbst betreibbar. Die großen Spitzenmodelle benötigen jedoch erhebliche Mengen Arbeitsspeicher, Massenspeicher und GPU-Leistung. Vollständige Varianten sind für einen gewöhnlichen Mac meist nicht praktikabel.

Für lokale Tests eignen sich eher:

- kleinere Coding-Modelle
- quantisierte Modelle
- Modelle mit bestätigtem Tool Calling
- ein interner GPU-Server
- ein OpenAI-kompatibler Unternehmensendpunkt

Kimi Code muss nicht zwingend ein Kimi-Modell verwenden. Der Agent kann auch mit kompatiblen lokalen Modellen anderer Hersteller arbeiten. Umgekehrt gilt: Ein lokales Chatmodell ist nicht automatisch ein guter Coding-Agent. Entscheidend ist, ob Tool Calling zuverlässig funktioniert, nicht die reine Chatqualität. Kontextgröße und Geschwindigkeit hängen dabei direkt von der verfügbaren Hardware ab.

Vor- und Nachteile

Vorteile:

- Quellcode kann innerhalb der eigenen Infrastruktur bleiben
- keine Tokenkosten
- Offlinebetrieb grundsätzlich möglich

- Kontrolle über Modell, Version und Server
- keine externen Rate Limits

Nachteile:

- hoher Hardware- und Speicherbedarf
- oft langsamere Verarbeitung
- kleinere Modelle sind weniger zuverlässig
- Tool Calling kann fehleranfällig sein
- Einrichtung und Wartung liegen beim Benutzer
- Kontextgröße hängt vom verfügbaren Speicher ab

18 Preise

Vorübergehende Aufnahmepause (Stand 20. Juli 2026)

Moonshot AI hat am 19. Juli 2026 mitgeteilt, dass die Nachfrage nach Kimi K3 innerhalb von 48 Stunden die verfügbare GPU-Kapazität überschritten hat. Seither sind neue Mitgliedschafts-Anmeldungen pausiert, sowohl für die allgemeine Kimi-Mitgliedschaft als auch für die Kimi-Code-Mitgliedschaft. Bestehende Abonnements bleiben laut Anbieter unverändert bestehen. Gleichzeitig wurde angekündigt, die Mitgliedschaft künftig in zwei getrennte Pläne aufzuteilen, eine allgemeine Kimi-Mitgliedschaft und eine eigene Kimi-Code-Mitgliedschaft, um Rechenkapazität für Coding-Workflows separat planen zu können. Ein Termin für die Wiedereröffnung wurde nicht genannt.

Diese Information stützt sich auf die offizielle Ankündigung von Moonshot AI sowie unabhängige Presseberichterstattung vom 18. bis 20. Juli 2026 und ist nicht vollständig über eine statische Dokumentationsseite nachprüfbar, da die Live-Preisseite als JavaScript-Anwendung ausgeliefert wird. Ob die Pause alle fünf Tarife gleichermaßen betrifft oder differenziert ist, ist nicht eindeutig belegt. Das gilt daher als **nicht offiziell im Detail bestätigt**. Vor einer Kaufentscheidung sollte der aktuelle Status direkt auf der Kimi-Preisseite geprüft werden. Die verbrauchsabhängige Kimi Platform API scheint von dieser Pause nicht betroffen zu sein.

Mitgliedschaft

Die folgenden Tarifdaten stammen von der offiziellen Kimi-Hilfeseite zur Mitgliedschaft, geprüft am 20. Juli 2026. Sie gelten für bestehende Abonnements unverändert; ob sie für neue Anmeldungen aktuell buchbar sind, hängt von der oben beschriebenen Aufnahmepause ab.

Tarif	Monatlich	Jährliche Zahlung	Kimi-Code-Kontingent
Adagio	0 US-Dollar	kostenlos	kein reguläres Kimi-Code-Kontingent
Moderato	19 US-Dollar	180 US-Dollar	1×
Allegretto	39 US-Dollar	372 US-Dollar	5×
Allegro	99 US-Dollar	948 US-Dollar	15×
Vivace	199 US-Dollar	1.908 US-Dollar	30×

Das Kimi-Code-Kontingent stammt aus einem eigenen Credit-Pool, getrennt vom allgemeinen Agent-Credit-Pool der Kimi-Mitgliedschaft. Ab Allegretto ist laut offiziellem Kimi-Code-Änderungsprotokoll zusätzlich die HighSpeed-Variante des Coding-Modells freigeschaltet.

Zusätzlich existiert „Extra Usage“, ein aufladbares Zusatzguthaben, das ausschließlich zahlende Mitglieder aktivieren können und das greift, sobald das reguläre Kontingent aufgebraucht ist, ohne die normale Kontingent-Erneuerung zu beeinflussen. Ein verlässlicher USD-Preis pro Einheit ließ sich nicht auf einer internationalen, englischsprachigen Preisseite verifizieren; auffindbare Beispielwerte stammen von der chinesischen Hilfeseite (in CNY) und sind **nicht offiziell für die internationale USD-Abrechnung bestätigt**.

Ob die genannten Preise Steuern enthalten, ist auf der Membership-Seite nicht angegeben.

API-Abrechnung (Kimi Platform)

Die Preise verstehen sich pro eine Million Tokens. Laut offizieller Preisseite gilt: „Prices exclude applicable taxes. Specific tax obligations are subject to local tax regulations and will be calculated at checkout based on your jurisdiction.“ Die Preise sind also explizit **vor Steuern** zu verstehen.

Modell	Kontextfenster	Eingabe mit Cache-Treffer	Eingabe ohne Cache-Treffer	Ausgabe
Kimi K2.7 Code (`kimi-k2.7-code`)	262.144 Token	0,19 US-Dollar	0,95 US-Dollar	4,00 US-Dollar
Kimi K2.7 Code HighSpeed (`kimi-k2.7-code-highspeed`)	262.144 Token	0,38 US-Dollar	1,90 US-Dollar	8,00 US-Dollar
Kimi K2.6 (`kimi-k2.6`, ältere, weiterhin angebotene Variante)	262.144 Token	0,16 US-Dollar	0,95 US-Dollar	4,00 US-Dollar
Kimi K3 (`kimi-k3`)	1.048.576 Token	0,30 US-Dollar	3,00 US-Dollar	15,00 US-Dollar

Bei K3 zählen Reasoning-Tokens laut Dokumentation als Output-Tokens und werden entsprechend zum Output-Preis abgerechnet.

Rate Limits richten sich nach dem kumulierten Aufladebetrag des Kontos und sind in sechs Stufen (Tier0 bis Tier5) gestaffelt: von 1 gleichzeitiger Anfrage und 500.000 Token/Minute auf der niedrigsten Stufe bis zu 1.000 gleichzeitigen Anfragen und 5.000.000 Token/Minute auf der höchsten Stufe. Modellspezifische Abweichungen einzelner Limits für K3 oder K2.7 Code waren in den geprüften Quellen nicht gesondert aufgeführt.

Ein Agent benötigt wegen wiederholter Dateioperationen, Tool-Aufrufe und Verarbeitungsschritte deutlich mehr Tokens als ein normaler Chat.

Der lokale Betrieb verursacht keine Tokenkosten. Dafür entstehen Hardwarebedarf, Stromverbrauch und Wartungsaufwand.

Unbestätigter Zusatzhinweis: Nach Sekundärquellen sollen ältere Modelle wie `kimi-k2.5` und die Moonshot-V1-Reihe zum 31. August 2026 vollständig abgeschaltet werden. Das ließ sich nicht direkt auf einer offiziellen Preis- oder Änderungsseite bestätigen und ist daher als vorläufig zu behandeln.

19 Datenschutz

Cloudbetrieb

Bei einem Cloudmodell werden die für eine Aufgabe benötigten Informationen an den Modellanbieter übertragen. Das gilt unabhängig davon, ob es sich um die Kimi-Code-Mitgliedschaft, die Kimi Platform API oder einen anderen Cloudanbieter handelt. Vor dem Einsatz mit Firmenprojekten müssen Datenschutz, Speicherung, Aufbewahrungsfristen, Trainingsnutzung und zulässige Datenarten geprüft werden.

Lokaler Betrieb

Ein lokaler Modellendpunkt garantiert allein noch keinen vollständig lokalen Arbeitsablauf. „Alles bleibt lokal“ stimmt nur, wenn tatsächlich sämtliche folgenden Komponenten deaktiviert oder ebenfalls lokal sind. Sonst führt Kimi Code trotz lokalem Modell weiterhin Netzwerkzugriffe aus:

- Websuche
- URL-Abrufe
- externe MCP-Server
- Plugins mit Cloudzugriff
- externe Build- oder CI-Dienste
- Telemetrie

Die Telemetrie lässt sich in `~/.kimi-code/config.toml` deaktivieren:

```
telemetry = false
```

Für einen kontrolliert isolierten Offlinebetrieb müssen zusätzlich externe MCP-Server, Webwerkzeuge und netzwerkfähige Plugins deaktiviert werden.

20 Migration von Claude Code

Für den Umstieg von Claude Code steht ein eingebauter Skill-Command zur Verfügung:

```
/import-from-cc-codex
```

Der Befehl importiert vorhandene Claude-Code- und Codex-Konfigurationen in die Kimi-Code-Struktur. Was sich dabei sinnvoll übertragen lässt:

Claude Code	Übertragung nach Kimi Code
`CLAUDE.md`	Inhalt lässt sich in `AGENTS.md` übernehmen
`.claude/` (Projektverzeichnis)	Entsprechende Inhalte wandern konzeptionell nach `.kimi-code/`
Skills (`.claude/skills/`)	`SKILL.md`-Dateien sind strukturell sehr ähnlich, meist mit kleinen Anpassungen übernehmbar
Custom Commands (`.claude/commands/`)	Kein direktes Äquivalent – müssen als Skill oder Plugin-Command neu abgebildet werden
MCP-Konfiguration (`.mcp.json`)	Serverdefinitionen lassen sich inhaltlich nach `.kimi-code/mcp.json` übertragen; Dateiname und -struktur unterscheiden sich
Hooks (`settings.json`)	Müssen manuell in `[[hooks]]`-Einträge in `config.toml` übersetzt werden – anderes Dateiformat (JSON statt TOML) und teils andere Event-Namen
Projektregeln allgemein	Inhaltlich meist direkt verwendbar, da beide Systeme auf Markdown-Anweisungsdateien setzen

Was `/import-from-cc-codex` im Detail liest, übernimmt und wie mit Namenskonflikten umgegangen wird, ist nicht bis ins letzte Detail primärquellenseitig dokumentiert. Es empfiehlt sich deshalb, das Ergebnis nach dem Import manuell zu prüfen, statt sich blind darauf zu verlassen.

Nicht automatisch oder nicht vollständig migrierbar:

- **Hooks** – unterschiedliches Konfigurationsformat (JSON vs. TOML) und teils unterschiedliche Event-Namen; eine automatische 1:1-Übersetzung ist nicht zu erwarten.
- **Custom Commands aus `.claude/commands/`** – Kimi Code kennt dieses Verzeichnis nicht; die Commands müssen als Skill oder Plugin-Command neu erstellt werden.
- **Plugins** – Claude-Code-Plugins (Marketplace-Konzept, `.claude-plugin/plugin.json`) sind nicht mit Kimi-Code-Plugins (`kimi.plugin.json`) kompatibel und müssen für Kimi Code neu gebaut werden.
- **Berechtigungsregeln** – die Permission-Modi unterscheiden sich (Claude Code: `default`, `acceptEdits`, `plan`, `auto`, `dontAsk`, `bypassPermissions`; Kimi Code: `manual`, `yolo`, `auto`) und lassen sich nicht 1:1 übertragen. Freigaberegeln sollten nach der Migration erneut geprüft werden.

- **Subagenten** – Definitionsformat und verfügbare Typen unterscheiden sich strukturell (Kimi Code: `coder`/`explore`/`plan` plus Swarm-Modus; Claude Code: frei definierbare Agents unter `.claude/agents/`).

Nach jeder Migration gilt: importierte Projektanweisungen, Skills, MCP-Einstellungen und Hooks vor der produktiven Nutzung manuell durchsehen.

21 Empfohlener Einstieg

Für einen ersten Test eignet sich ein kleines Git-Projekt mit vorhandenen Unit Tests.

8. Kimi Code installieren.
9. Im Projekt einen separaten Branch erstellen.
10. Mit `/login` ein Cloudmodell oder über `/provider` ein lokales Modell einrichten.
11. Mit `/init` eine `AGENTS.md` erzeugen.
12. Die erzeugten Projektanweisungen manuell korrigieren.
13. Im Plan-Modus eine kleine Aufgabe analysieren lassen.
14. Nur einen überschaubaren Teil umsetzen lassen.
15. Build, Tests und Git-Diff kontrollieren.
16. Erst danach größere Aufgaben oder autonomere Modi (Goal, Swarm) ausprobieren.

Wer von Claude Code umsteigt, findet im Kapitel „Migration von Claude Code“ eine Übersicht, was sich automatisiert übernehmen lässt und was manuell nachgezogen werden muss.

22 Quellen

Letzte inhaltliche Prüfung: 20. Juli 2026

Kimi Code

- Kimi Code Homepage
- Getting Started
- Migration von der alten CLI
- Goals (Goal-Modus)
- Konfigurationsdateien
- Datenverzeichnisse
- Provider-Konfiguration
- Umgebungsvariablen
- Agents und Subagenten
- Skills
- Hooks
- Plugins
- MCP
- Slash Commands
- kimi-Command-Referenz
- Tools-Referenz
- Kimi-Code-Änderungsprotokoll
- Kimi Code auf GitHub
- Alte kimi-cli auf GitHub (Referenz, wird ausgemustert)

Kimi Platform und Preise

- Kimi Platform
- Kimi Platform Dokumentation
- Kimi K3 API-Preise
- Kimi K2.7 Code API-Preise
- Kimi K2.6 API-Preise
- Rate Limits
- Kimi-Mitgliedschaften und Preise

Claude Code

- Claude Code Dokumentation (aktuell unter code.claude.com; docs.anthropic.com/en/docs/claude-code leitet dorthin weiter)
- CLI-Referenz
- Permission-Modi
- Settings
- Memory / CLAUDE.md
- Skills
- Hooks
- Plugins
- MCP
- Subagents
- Sessions
- Headless-Modus / Automatisierung
- GitHub-Actions-Integration

LM Studio und Ollama

- LM Studio Dokumentation
- LM Studio OpenAI-Kompatibilität
- LM Studio Tool Calling
- LM Studio lokaler Server
- LM Studio Authentifizierung
- LM Studio CLI (`lms`)
- Ollama Dokumentation
- Ollama OpenAI-Kompatibilität
- Ollama Tool Calling
- Ollama Kontextgröße

23 Änderungsprotokoll

- Fließtext von der klassischen KI-typischen Häufung des Gedankenstrichs als Einschub-Zeichen entlastet (rund 20 Stellen), meist ersetzt durch Punkt, Komma, Doppelpunkt oder Klammer – je nachdem, was für den jeweiligen Satz am natürlichsten klang.
- Eine negative Parallelismus-Konstruktion aufgelöst („nicht nur … sondern auch“ in der Einleitung des Vergleichskapitels).
- Listen, Tabellen, Codeblöcke, Überschriften und alle inhaltlichen Aussagen unverändert gelassen – hier ging es ausschließlich um den Sprachstil, nicht um neue Recherche oder Faktenänderungen.
- Das Original `kimi-code-de.md` wurde dabei nicht angetastet; diese Datei ist eine eigenständige Kopie.