KI für größere Projekte nutzen

- Große Code-Projekte mit KI bauen: Ein vollständiger Leitfaden
- Agenten?
- TTS 2026: OmniVoice, Qwen3-TTS, Chatterbox, Kokoro, NeuTTS, VoxCPM2 und Gepard 1.0 im Vergleich

Große Code-Projekte mit KI bauen: Ein vollständiger Leitfaden □□

Die zentrale Herausforderung, die du beschreibst, ist real: LLMs haben endliche Kontextfenster, keine persistente Erinnerung zwischen einzelnen Aufrufen und werden unzuverlässiger, sobald Aufgaben im Umfang wachsen. Große Projekte erfordern, dass du eine **System-Umgebung um das Modell herum** baust – statt dich nur auf das Modell allein zu verlassen. Unten ist eine detaillierte Vorgehensweise.

---

1. Das Kern-mentale Modell

Denk die KI nicht als Programmierer, der sich an alles erinnert, sondern als einen **extrem fähigen Auftragnehmer mit kompletter Amnesie**, der jeden Morgen frisch startet. Jeder Prompt ist ein neuer Arbeitstag. Alles, was die KI wissen muss, muss entweder:

1. **Im Prompt** (eingespeister Kontext) stecken, oder
2. **Auffindbar** sein (Dateien, die sie lesen kann, Tools, die sie aufrufen kann).

Deine gesamte Aufgabe ist *Context Engineering*: sicherzustellen, dass die richtige Information zur richtigen Zeit da ist und die Ergebnisse außerhalb des Modells persistent gespeichert werden.

Die drei Feinde großer Projekte sind:

- **Kontextfenster-Limits** — du kannst keine 200k-Zeilen-Codebasis einfügen.
- **Context Rot** — Modellqualität sinkt, sobald der Kontext sich füllt, sogar deutlich *unterhalb* des harten Limits. Zuverlässigkeit nimmt oft schon ab, bevor du die Max-Tokens erreichst.
- **State Loss** — das Modell vergisst Entscheidungen, die 10 Prompts zuvor gefallen sind.

Alles Folgende ist darauf ausgelegt, genau diese drei Probleme zu bekämpfen.

---

2. Die Grundlage: Spezifikationsgetriebene Entwicklung

Starte nie damit, die KI zu bitten: „Bau mir eine App.“ Starte damit, die KI zu nutzen, um **dauerhafte Artefakte** zu erzeugen, die in deinem Repository leben und über Sitzungen hinweg überdauern.

Die Dokument-Hierarchie

Erstelle diese Dateien *bevor* du Production Code schreibst:

Datei	Zweck
`SPEC.md` / `PRD.md`	Was du baust und warum. Anforderungen, User Stories, Constraints.
`ARCHITECTURE.md`	High-Level-Design: Komponenten, Datenfluss, Tech-Stack, zentrale Entscheidungen.
`SCHEMA.sql` / `types.ts`	Datenmodelle — die gemeinsame Sprache des ganzen Projekts.
`TASKS.md` / `PLAN.md`	Zerlegte Task-Liste mit Status-Tracking.
`CONVENTIONS.md`	Coding Standards, Patterns, Benennung, Ordnerstruktur.
`DECISIONS.md`	Ein append-only Log architektureller Entscheidungen (ADRs) und *warum*.

Du erstellst diese *mit* der KI in einer interaktiven Planungsphase. Danach werden sie zur **Single Source of Truth**, die du in zukünftigen Prompts wieder einspielst. Das ist die Praxis mit dem höchsten Hebelwirkungseffekt für große Projekte.

Interaktive Planungs-Technik

Nutze ein starkes Reasoning-Modell (Opus, GPT-5.5) im „Architect Mode“. Fordere es explizit auf, dich zu „verhören“:

> „Du bist ein Senior Architect. Bevor du irgendeinen Code schreibst, stelle mir so lange klärende Fragen, bis es null Unklarheit über die Anforderungen gibt. Dann erstelle ARCHITECTURE.md und eine abhängigkeitsorientierte Task-Aufteilung. Schreibe noch keinen Implementierungs-Code.“

Eine **separate Plan-/Architect-Phase vom Code-Teil** zu erzwingen, ist eine der effektivsten Techniken überhaupt. Modelle liefern deutlich besseren Code, wenn sie zuerst über Struktur nachgedacht haben.

---

3. Zerlegung: Ein großes Projekt in kleine Prompts verwandeln

Der Kern der Technik. Du musst das Projekt in Einheiten schneiden, die in *einen* Prompt passen — und dabei noch **Luft lassen**.

Prinzipien guter Zerlegung

- **Vertikale Slices statt horizontale Layers.** Bevorzuge „User kann end-to-end einloggen“ statt „erst komplette Datenbank bauen, dann komplette API“. Vertikale Slices sind unabhängig testbar und liefern früh funktionierende Software.
- **Jede Task braucht ein klares, überprüfbares Completion-Kriterium.** „Implementiere den `POST /users`-Endpoint mit Validierung und Tests“ — nicht „arbeite am User-System“.
- **Respektiere den Dependency Graph.** Erst Interfaces und Datenmodelle, dann Implementierungen, die davon abhängen. Die KI kann nicht korrekt einen Consumer einer API implementieren, die es noch nicht gibt.
- **Ziel: grob 200–600 Zeilen Änderung pro Task.** Klein genug zum Review, groß genug, um sinnvoll zu sein.

Das Interface-First Pattern

Definiere Contracts, bevor du implementierst. Lass die KI zuerst alle **Type Signatures / Interfaces / API-Schemas** als Stubs generieren, commite sie, und implementiere dann jeden Stub in separaten Prompts. Da die Interfaces in Dateien „eingefroren“ sind, kann jeder nachfolgende Prompt sie lesen und konsistent bleiben — auch wenn das Modell selbst keine Erinnerung an das Schreiben hat.

```
Prompt 1: Define all interfaces/types (thin, no logic)  → commit
Prompt 2: Implement module A against interfaces         → test → commit
Prompt 3: Implement module B against interfaces         → test → commit
...
```

Das entkoppelt die Tasks so, dass jede nur die Interfaces plus ihr eigenes Modul sehen muss — nicht die komplette Codebase.

---

4. Kontext über Prompts hinweg managen

Hier scheitern die meisten. Konkrete Strategien:

A. Selektives Context Injection

Nie das ganze Repo auskippen. Gib pro Task nur das mit:

- den relevanten Ausschnitt von `ARCHITECTURE.md` und `CONVENTIONS.md`
- die spezifischen Interfaces/Typen, die die Task berührt
- die 1–3 Dateien, die geändert werden
- Beispiele für *ähnlichen* bereits existierenden Code (damit das Modell deinen Stil trifft)

B. Repository Maps

Zur Orientierung ohne Volltext: Gib dem Modell einen **Dateibaum plus einzeilige Zusammenfassungen** jeder Datei — oder nur die Funktions-Signaturen (eine „Repo Map“). So arbeiten Tools wie Aider: Sie bauen eine komprimierte Karte aus der Code-Struktur, damit das Modell weiß, was existiert, und gezielt den kompletten Inhalt nur der benötigten Stellen anfordern kann.

C. Summarisierung / Rolling Memory

Wenn ein Gespräch lang wird, lass das Modell den **aktuellen Session-Status** in ein Handoff-Dokument verdichten, bevor der Kontext voll ist:

> „Fasse alles, was in dieser Session entschieden und erledigt wurde, in einem HANDOFF.md zusammen, das eine frische Instanz nutzen kann, um fortzusetzen. Enthalten sein müssen: aktuelle Dateizustände, offene Fragen und nächste Schritte.“

Dann starte ein neues Gespräch, das mit dieser Zusammenfassung „geseedet“ wird. Das ist manuelles „Context Compaction“.

D. Retrieval (RAG) für sehr große Codebases

Bei wirklich riesigen Projekten: indexiere die Codebase in einen Vector Store (Embeddings) und hole pro Query semantisch relevante Code-Chunks. Das ist mehr Setup, aber es erlaubt dem Modell, passenden Code „zu finden“, den es im Kontext nie gesehen hat. Viele Agent-Frameworks machen das automatisch.

---

5. Die Agentische Schleife (Der echte Unlock) □□

Der größte Sprung bei großen Projekten ist, von **Chat** zu **Agents** zu wechseln — gib dem Modell Tools, damit es autonom über viele interne Schritte handeln kann, ohne dass du Copy-Paste machen musst.

Eine Agent läuft eine Schleife:

```
1. Read task + relevant files (tool: read_file, search)
2. Reason about approach
3. Make edits (tool: write_file / apply_diff)
4. Run tests / build (tool: execute_shell)
5. Read the output; if failing, go to 2
6. Repeat until task's completion criterion is met
7. Commit (tool: git)
```

Die entscheidende Erkenntnis: **Das Feedback-Loop von echtem Tool-Output (Compiler-Fehler, Test-Fehlschläge, Runtime-Logs) macht KI in großem Maßstab zuverlässig.** Ein Modell, das Tests ausführen kann und sieht, dass sie fehlschlagen, korrigiert seine eigenen Fehler.

Ein Modell, das nur in eine Leere schreibt, sammelt Fehler.

Tools, die du einem Agent mindestens geben solltest

- Datei lesen/schreiben/listen/suchen (grep)
- Shell-Ausführung (build, tests, linters)
- Version Control (git diff, commit)
- Optional: Websuche, Docs-Lookup

Fertige Agent-Harnesses

Statt alles selbst zu bauen, nutze existierende Tools, die mit OpenAI-kompatiblen Endpoints funktionieren:

- **Aider** — terminalbasiert, sehr stark beim Repo-Mapping, git-integriert, model-unabhängig.
- **Cline / Roo Code** — VS-Code-Extensions mit vollständigen agentischen Loops, zeig sie auf deine Custom Base URL.
- **OpenHands (ehemals OpenDevin)** — autonomer Software-Engineering-Agent.
- **Continue.dev** — konfigurierbar, unterstützt Custom OpenAI-kompatible Provider.
- **Claude Code / Codex CLI** — falls du sie auf deinen Endpoint routen kannst.

Alle nehmen ein benutzerdefiniertes `base_url` + `api_key`, sodass dein Multi-Model-Gateway direkt „reinschlägt“.

---

6. Multi-Model Orchestrierung (Nutze deine Setup-Stärke)

Da du viele Modelle hinter einer einzigen API hast, nutze ihre unterschiedlichen Stärken. Verwende nicht ein Modell für alles.

Rolle	Best-fit Modell	Warum
Architect / Planner	Opus, GPT-5.5 (High Reasoning)	Tiefe Argumentation, sieht das ganze Bild, die Kosten lohnen sich fürs Planen.

Rolle	Best-fit Modell	Warum
Implementer	Claude Sonnet, Mid-tier GPT	Schnell, günstig, stark bei gut spezifizierten Coding Tasks. Der Großteil eures Volumens.
Reviewer / Critic	Ein *anderes* starkes Modell	Neue Perspektive findet Bugs; nutze ein anderes Modell als den Autor, um geteilte Blind Spots zu vermeiden.
Günstige Hilfsarbeiten	Kleinstes fähiges Modell	Renaming, Boilerplate, Docstrings, Test-Scaffolding.

Starke Multi-Model Patterns

- **Generator–Critic Loop:** Modell A schreibt Code, Modell B reviewt gegen Spezifikation und Tests, Modell A überarbeitet. Cross-Model-Review ist erstaunlich effektiv, um Fehler zu finden, die ein einzelnes Modell wegargumentiert.
- **Plan then execute split:** Teures Reasoning-Modell erzeugt einen detaillierten Step-Plan; günstiges Modell führt jeden Step aus. Große Kosteneinsparung.
- **Ensemble / best-of-N:** Für eine harte, kritische Funktion: 2–3 Modelle lösen sie jeweils, dann wählt/synthetisiert ein Judge-Modell das beste Ergebnis. Teuer; nur für wirklich harte Probleme.
- **Escalation:** Starte mit einem günstigen Modell; wenn es bei N Versuchen fehlschlägt, eskaliere die Task zu einem stärkeren Modell – mit den Failure-Logs als Anlage.

---

7. Verifikation: Der nicht verhandelbare Backbone

AI-Code ist mit hoher Selbstsicherheit falsch — und die Fehlerrate kannst du über tausende Zeilen nicht ignorieren. Dein Sicherheitsnetz ist **automatisierte Verifikation**, und die muss engmaschig sein.

Baue ein „Harness“, auf das die KI sich stützen kann

- **Tests zuerst (oder früh).** Lass die KI Tests zusammen mit Code schreiben. Noch besser: definiere Acceptance Tests für eine Task *vor* der Implementierung, sodass das Completion-Kriterium wörtlich „diese Tests sind grün“ ist.

- **Types & Linting.** Nutze eine statisch typisierte Sprache oder striktes Type Checking (TypeScript strict, mypy, etc.). Der Type Checker ist ein kostenloser, ausdauernder Reviewer, der ganze Klassen von KI-Halluzinationen einfängt (z. B. Funktionen, die es nicht gibt, falsche Signaturen).
- **CI Gates.** Jeder AI-Commit läuft die komplette Suite. Nichts merged rot.
- **Kleine, reviewbare Diffs.** Weil du in kleine Tasks zerlegt hast, sind die Diffs menschlich reviewbar. **Du musst den Code trotzdem lesen.** Behandle AI-Ausgaben als PR eines talentierten, aber nicht verantwortlichen Junior-Dev.

Je enger deine Feedback-Loop, desto mehr Autonomie kannst du der KI sicher geben. Projekte mit guter Testabdeckung lassen sich viel stärker automatisieren als solche ohne.

---

8. State & Progress Tracking

Da Sessions stateless sind, musst du Fortschritt externisieren.

- Halte `TASKS.md` mit Checkbox-Status (`[ ]`, `[x]`, `[blocked]`). Update es am Ende jeder Task. Spiele es zu Beginn jeder Task ein, damit das Modell weiß, wo ihr steht.
- Nutze **git commits pro Task** mit aussagekräftigen Messages — dein git log wird dadurch zu einer dauerhaften, abfragbaren Projekt-Historie.
- Pflege das append-only `DECISIONS.md`, damit die KI nicht erneut ungelöste Entscheidungen verhandelt oder frühere Designs widerspricht.
- Erwäge eine `CLAUDE.md` / `AGENTS.md` Datei im Repo-Root — viele Agent-Tools laden das automatisch als persistente Projektanweisungen (Konventionen, Commands zum Testen, typische Fallstricke). Das gibt jeder neuen Session das gleiche Baseline-Briefing.

---

9. Ein konkreter End-to-End Workflow

So passt das alles in ein reales Projekt zusammen:

Phase 0 — Planning (starkes Reasoning-Modell, Chat-Modus)

1. Interaktives Q&A → erstelle `SPEC.md`.
2. → erstelle `ARCHITECTURE.md` und Datenmodelle.
3. → dekomponiere in abhängigkeitsorientierte `TASKS.md`.
4. Richte Repo, CI, Test-Framework, `AGENTS.md` ein.

Phase 1 — Scaffolding (Mid-Modell)
5. Generiere Projektskelett, Config, alle Interface/Type Stubs. Committen.

Phase 2 — Iterative Implementierung (agentische Loop, cheap→mid Modell)
Für jede Task in Dependency-Reihenfolge:
6. Frischer Kontext: konventionen + relevante Interfaces + Ziel-Dateien + Task.
7. Agent schreibt Code + Tests, führt sie aus, iteriert bis „grün“.
8. Reviewer-Modell kritisiert das Diff gegen die Spezifikation.
9. Author-Modell adressiert das Review.
10. Menschliches Review des Diffs, merge, Update `TASKS.md`. Committen.

Phase 3 — Integration & Hardening
11. Integrationstests über Slices hinweg.
12. Modelle lassen auf Security, Performance, Edge Cases reviewen.
13. Refactoring-Passes (jetzt, wo die Form klar ist).

Phase 4 — Wartung
14. Gleiche Loop für jede neue Feature/Bugfix; die Artefakte halten das Projekt auf Dauer kohärent.

---

10. Praktische Fallstricke & Regeln für Daumen

- **Lass den Kontext nicht über ~50–70% wachsen.** Qualität sinkt, bevor du das harte Limit erreichst. Kompakte oder starte früher neu, als du denkst.
- **Eine Task pro Conversation**, wenn die Arbeit komplex ist, um Cross-Contamination und Kontextaufblähung zu vermeiden.
- **Interfaces einfrieren, bevor du parallelisierst.** Ein Shared Type „mittendrin“ zu ändern zwingt alles zur Neuarbeit.
- **Lass die KI ihren Plan zeigen, bevor sie irgendetwas Nicht-Triviales editiert** („Erkläre deinen Ansatz, liste die Dateien, die du ändern wirst, dann warte auf mein Go“). Günstige Versicherung gegen 400 Zeilen in die falsche Richtung.
- **Achte auf silent scope drift** — das Modell könnte „hilfreich“ Dinge umschreiben, die du nicht angefordert hast. Begrenze es: „Ändere nur X; ändere keinen verwandten/unbeteiligten Code.“
- **Regenereiere statt Patch** für kleine Dateien; **surgical diffs** für große. Diff/Patch Tools sparen Tokens und reduzieren Fehler bei großen Dateien.
- **Versioniere auch Prompts/Artefakte in git** — dein Prompting ist jetzt Teil deines Engineering-Prozesses.
- **Cost Control:** Routings nach Task-Schwierigkeit, statischen Kontext cachen (system prompt, Konventionen), falls deine API Prompt Caching unterstützt, und Diffs statt Full-File-Rewrites bevorzugen.

Zusammenfassung

Große, mit KI gebaute Projekte gelingen nicht, weil das Modell schlau genug ist, um alles im Kopf zu halten — das kann es nicht — sondern weil du ein **diszipliniertes System** baust: dauerhafte Spezifikations-Artefakte als Single Source of Truth, aggressive Zerlegung in überprüfbare Tasks, eine agentische Loop auf Basis echter Test-/Build-Feedbacks, selektives Context Injection und Multi-Model-Orchestrierung, die jedes Modell auf seine Stärke ausrichtet. Die KI liefert die rohe Fähigkeit; **du lieferst das Gedächtnis, die Struktur und die Verifikation.** Wenn diese drei stimmen, gibt es praktisch keine harte Grenze für die Projektgröße. □□

Agenten?

Basierend auf der aktuellen Recherche (Mitte 2026) sieht die Landschaft so aus — speziell gefiltert für **deine Situation**, denn genau das ist entscheidend: Du hast einen einzelnen **OpenAI-kompatiblen Endpunkt mit vielen Modellen dahinter**. Diese Einschränkung schließt viele sonst sehr gute Tools aus oder stuft sie herunter und hebt die „Bring-your-own-model“-Tools (BYOM) nach oben. □□

Zuerst die wichtigste Einordnung: Agent = Modell + Gerüst

Die wichtigste Erkenntnis aus der aktuellen Recherche ist, dass **das Gerüst genauso wichtig ist wie das Modell**. In einem Test vom Februar 2026 erzielte *dasselbe* Modell (Opus 4.5), das über drei verschiedene Agenten-Frameworks lief, einen Unterschied von 17 Problemen in einem Datensatz mit 731 Problemen — ein Abstand so groß wie eine komplette Modellgeneration. Deine Wahl des *Gerüsts* ist also nicht nur Kosmetik, sondern bestimmt direkt die Qualität der Ergebnisse.

Da du die Modelle selbst bereitstellst, lautet deine eigentliche Entscheidung also: **Welches Gerüst nutzt einen benutzerdefinierten OpenAI-kompatiblen Endpunkt mit mehreren Modellen am besten aus?**

Das gesamte Feld, nach Eignung für dein Setup gruppiert

Stufe A — Beste Wahl für dich (native BYOM, Multi-Modell, OpenAI-kompatibel)

Diese Tools lassen dich auf eine benutzerdefinierte `base_url` zeigen, deinen Schlüssel eintragen und verschiedene Modelle verschiedenen Rollen zuweisen. Genau das ist dein Anwendungsfall.

Tool	Formfaktor	Warum es zu dir passt
Cline	VS-Code-Erweiterung	Ca. 5 Mio. Installationen; explizite Konfiguration für „OpenAI Compatible“ (Basis-URL + Schlüssel + Modell-IDs). Der Plan-/Act-Modus trennt Architekt- und Coder-Rollen — perfekt, um Opus für die Planung und Sonnet für die Ausführung zuzuweisen. Kein Inferenz-Aufschlag.
Roo Code	VS-Code-Erweiterung (Cline-Fork)	Hat den Ruf, **bei großen Änderungen über viele Dateien hinweg am zuverlässigsten** zu sein. Mehr Konfigurationsmöglichkeiten, benutzerdefinierte „Modi“ (Rollen), die du an bestimmte Modelle binden kannst. Klassenbeste Lösung für das von dir beschriebene „große Projekt“-Problem.
Aider	CLI / Git-nativ	Ausgezeichnetes Repo-Mapping, atomare Git-Commits, funktioniert mit jedem OpenAI-kompatiblen Modell. Integrierter „Architect Mode“, der ein starkes Reasoning-Modell für die Planung und ein günstigeres Modell zum Schreiben des Diffs nutzt — also genau das Multi-Modell-Muster, das ich vorher beschrieben habe, sofort einsatzbereit.
OpenHands (früher OpenDevin)	Selbst gehosteter autonomer Agent	MIT-lizenziert, über 100 Backends, jede OpenAI-kompatible API. Führt einen vollständigen CodeAct-Zyklus aus (Code ausführen, Tests laufen lassen, browsen) in einer Docker-Sandbox. 72 % auf SWE-bench Verified. Die autonomste Option unter den BYOM-Tools.
Kilo Code	VS-Code-Erweiterung	Holt schnell auf; fokussiert auf enge Kontextkontrolle und strukturierte Modi; dokumentierte Unterstützung für benutzerdefinierte OpenAI-kompatible Provider.

Stufe B — Hervorragende Tools, aber mit deinem Endpunkt etwas umständlich

- **Claude Code** — derzeit der **Spitzenreiter bei der Codequalität** (Opus 4.7/4.8), mit Selbstverifikation (schreibt Tests, führt sie aus, behebt Fehler vor der Antwort). Aber es ist

auf die eigene API bzw. das eigene Abo von Anthropic ausgelegt; es über ein beliebiges OpenAI-kompatibles Gateway zu leiten, ist eher ein Kampf als eine Funktion.
- **OpenAI Codex CLI** — aktuell **#1 auf Terminal-Bench** (GPT-5.5, 82,7 %), sehr stark für DevOps- und Terminal-Workflows. An OpenAIs API bzw. ChatGPT-Pläne gebunden; eine benutzerdefinierte Base-URL war historisch eher eine gewünschte, aber eingeschränkte Funktion.
- **Cursor** — die beliebteste KI-IDE, modellagnostisch *innerhalb ihrer eigenen unterstützten Liste* (Opus 4.7, GPT-5.5, Gemini 3.1), aber sie akzeptiert nicht sauber einen beliebigen Drittanbieter-Endpunkt. Außerdem nur als VS-Code-Fork.
- **GitHub Copilot** — inzwischen Multi-Modell und 2026 mit Base-URL-Override, aber eher auf Enterprise-/Compliance-Standards ausgerichtet und mit einem Abrechnungsmodell auf Basis von Nutzungsguthaben unterwegs. Weniger natürlich für ein reines Custom-Gateway-Setup.

Stufe C — Autonom / Nische

- **Devin 2.0** — vollständig isolierter Cloud-Ingenieur; stark bei *klar abgegrenzten* Aufgaben (Migrationen, Testabdeckung), schwächer bei mehrdeutigen Aufgaben. Kein BYOM.
- **Augment Code** — herausragende **Repository-Kontext-Engine** (indiziert das ganze Repo vor dem Start), über MCP nutzbar, sodass du die Indizierung zusammen mit einem anderen Generator verwenden könntest. Preislich eher Enterprise.

Meine Empfehlung für dich

Mit deinem OpenAI-kompatiblen Multi-Modell-Gateway und deinem Ziel von **großen Projekten, die über einen einzelnen Prompt hinausgehen**, würde ich einen **mehrschichtigen BYOM-Stack** aufsetzen:

1. Hauptwerkzeug: **Roo Code** (oder Cline)

Das ist dein wichtigstes Arbeitstool. Gründe:

- Nativer „OpenAI Compatible“-Provider — du kannst die URL und den Schlüssel deines Gateways direkt eintragen.
- **Benutzerdefinierte Modi, die an verschiedene Modelle gebunden sind** — das ist für dich die entscheidende Funktion. Konfiguriere:
  - **Architect/Plan-Modus → Opus** (oder GPT-5.5) für Zerlegung und Design.
  - **Code-Modus → Claude Sonnet** für den Großteil der Implementierung (schnell, günstig, stark).

    - **Debug/Review-Modus → ein *anderes* starkes Modell** als der Autor, für modellübergreifendes Review.
- Zuverlässig bei Änderungen über viele Dateien und mit langem Horizont — genau das, was große Projekte brauchen.
- Läuft in VS Code, sodass du Diffs, Review und den Loop eng integriert hast.

Nimm **Cline**, wenn du eine etwas einfachere, rundere Erfahrung möchtest; nimm **Roo Code**, wenn du maximale Kontrolle und Zuverlässigkeit bei großen Refactorings willst (das ist sein Haupt-Ruf).

2. Schwere Refactorings / CLI-Fans: **Aider**

Behalte Aider für Git-native Refactorings mit Commit-pro-Änderung im Werkzeugkasten. Sein **Architect + Editor-Zwei-Modell-Modus** passt perfekt zu deinem Multi-Modell-Zugang: Richte die Architekten-Rolle auf Opus und die Editor-Rolle auf Sonnet aus. Ideal, wenn Korrektheit und eine saubere Historie wichtiger sind als IDE-Komfort.

3. Vollautonome Ausführung: **OpenHands** (optional)

Wenn du klar definierte Aufgaben komplett autonom laufen lassen willst (Code schreiben → Tests ausführen → iterieren in einer Sandbox), dann hoste OpenHands selbst gegen deinen Endpunkt. Am besten für den Anwendungsfall „Lass es über Nacht an diesem Modul arbeiten“. □□□□

Modell-Routing, das du in allen Tools konfigurieren solltest

Rolle	Modell	Begründung
Architekt / Planer	**Opus** (oder GPT-5.5)	Tiefstes Reasoning; derzeit führend bei der Codequalität. Die Kosten lohnen sich für die Planung.
Implementierer (Masse)	**Claude Sonnet**	Schnell, günstig, stark beim Coden — deine Rolle mit dem höchsten Volumen.
Terminal-/DevOps-Aufgaben	**GPT-5.5**	#1 auf Terminal-Bench (82,7 %); am besten für Shell- und Pipeline-Arbeit.
Reviewer / Kritiker	Ein *anderes* Modell als der Autor	Modellübergreifendes Review erkennt gemeinsame blinde Flecken.

Zwei wichtige Hinweise aus der Recherche

1. **Benchmarks sind im Moment unsicher.** SWE-bench Verified wurde als kontaminiert eingestuft (Februar 2026) — jedes Frontier-Modell konnte die Gold-Antworten aus dem Gedächtnis reproduzieren. Behandle veröffentlichte Werte daher nur als *grobe Orientierung* und **lasse 50–100 Aufgaben aus deiner eigenen Codebasis** durch dein Kandidaten-Setup laufen, bevor du dich festlegst.
2. **Die Sicherheitslage unterscheidet sich deutlich.** BYOM-Tools wie Cline/Roo/Aider laufen standardmäßig mit **lokalem Systemzugriff** — sie können `.env`-Dateien lesen und Shell-Befehle ausführen. Für große autonome Projekte solltest du eine klare menschliche Review-Schranke definieren und über Sandboxing nachdenken (deshalb ist die Docker-Isolation von OpenHands für unbeaufsichtigte Läufe so attraktiv).

Kurz gesagt: Starte mit **Roo Code (oder Cline) als deinem primären Agenten**, konfiguriert mit modellbezogenem Routing pro Modus über dein Gateway; ergänze **Aider** für Git-native Refactorings; und setze optional **OpenHands** für autonome, isolierte Aufgabenausführung ein. Diese Kombination holt den größten Nutzen aus deiner OpenAI-kompatiblen Multi-Modell-API heraus und setzt direkt den Planen-→-Bauen-→-Reviewen-Loop um, den Projekte brauchen, die zu groß für einen einzelnen Prompt sind. □□

TTS 2026: OmniVoice, Qwen3-TTS, Chatterbox, Kokoro, NeuTTS, VoxCPM2 und Gepard 1.0 im Vergleich

Die Open-Source-Landschaft für Text-to-Speech (TTS) hat sich in den letzten 12 Monaten dramatisch verändert. Was früher nur mit teuren APIs wie ElevenLabs möglich war – Zero-Shot-Voice-Cloning, emotionale Kontrolle, Streaming in Echtzeit – gibt es heute frei verfügbar zum Download. In diesem Artikel werfen wir einen detaillierten Blick auf sieben der spannendsten aktuellen TTS-Modelle: **OmniVoice**, **Qwen3-TTS**, **Chatterbox**, **Kokoro**, **NeuTTS**, **VoxCPM2** und **Gepard 1.0**.

Kurzüberblick: Die Modelle auf einen Blick

Modell	Entwickler	Parameter	Lizenz	Kernstärke
OmniVoice	k2-fsa	– (Diffusion-LM-Architektur)	Apache 2.0	600+ Sprachen, breiteste Sprachabdeckung
Qwen3-TTS	Alibaba (Qwen-Team)	0.6B / 1.7B	Apache 2.0	Ultra-Low-Latency-Streaming, Instruction-Control
Chatterbox	Resemble AI	0.5B	MIT	Emotion-Exaggeration, schlägt ElevenLabs in Blindtests
Kokoro	hexgrad (Indie)	82M	Apache 2.0	Winziges, extrem effizientes Modell

Modell	Entwickler	Parameter	Lizenz	Kernstärke
NeuTTS (Air/Nano/2E)	Neuphonic	120–360M aktiv	Apache 2.0 / NeuTTS Open License	On-Device/Offline, läuft auf CPU/Raspberry Pi
VoxCPM2	OpenBMB	2B	Open Source	Tokenizer-freie Diffusion, 48kHz Studio-Qualität
Gepard 1.0	nineninesix	555M	Open Source	Streaming-first, ~50ms Time-to-First-Audio

---

1. OmniVoice – Der Sprachen-Champion

OmniVoice von k2-fsa (bekannt aus dem sherpa-onnx-Ökosystem) ist ein massiv multilinguales Zero-Shot-TTS-Modell und unterstützt beeindruckende **über 600 Sprachen** – das ist die derzeit breiteste Sprachabdeckung unter allen Zero-Shot-TTS-Systemen überhaupt.

Highlights:

- Basiert auf einer neuartigen **Diffusion-Language-Model-Architektur**
- Voice Cloning bereits mit 3–10 Sekunden Referenzaudio
- **Voice Design** über Attribute wie Geschlecht, Alter, Tonhöhe, Akzent/Dialekt (inkl. chinesischer Regionaldialekte wie Sichuanesisch) – ganz ohne Referenzaudio
- Feinsteuerung über Inline-Tags (`[laughter]`, `[sigh]`) und Aussprachekontrolle via Pinyin oder CMU-Phonemen
- Sehr schnelle Inferenz: RTF (Real-Time Factor) bis zu 0,025 – also **40× schneller als Echtzeit**
- Lizenz: Apache 2.0, auf GitHub bereits 8.500+ Stars

Einsatzgebiete: Ideal für Projekte, die tatsächlich globale Sprachabdeckung brauchen (Lokalisierung, Sprachassistenten für Nischensprachen), weniger für Ultra-Low-Latency-Dialoganwendungen.

---

2. Qwen3-TTS – Alibabas Streaming-Kraftpaket

Alibabas Qwen-Team hat mit **Qwen3-TTS** ein technisch besonders ausgereiftes Modell veröffentlicht, das in zwei Größen erscheint: **0.6B** und **1.7B** Parameter, jeweils in den Varianten Base (Voice Clone), CustomVoice (feste Premium-Stimmen) und VoiceDesign (Stimme per Text-Prompt erzeugen).

Highlights:

- Eigener Tokenizer **Qwen3-TTS-Tokenizer-12Hz** – erzielt in Benchmarks Bestwerte bei PESQ, STOI und UTMOS gegenüber Konkurrenten wie Mimi oder X-Codec2
- **Dual-Track-Streaming-Architektur**: Ein Modell für Streaming UND Nicht-Streaming, End-to-End-Latenz bis **97ms**
- Deckt 10 Hauptsprachen ab: Chinesisch, Englisch, Japanisch, Koreanisch, Deutsch, Französisch, Russisch, Portugiesisch, Spanisch, Italienisch
- In Benchmarks (Seed-TTS-Test) schlägt Qwen3-TTS-12Hz-1.7B-Base sogar CosyVoice3 und MiniMax-Speech bei der Wortfehlerrate (WER) im Englischen (1,24 vs. 1,45/1,65)
- Bei Sprecherähnlichkeit (SIM) und Instruction-Following liegt es klar vor ElevenLabs und GPT-4o-Audio
- Day-0-Support in vLLM-Omni für produktionsreife Bereitstellung
- Lizenz: Apache 2.0, 12.600+ GitHub-Stars

Einsatzgebiete: Sehr starke Allround-Wahl für mehrsprachige Sprachassistenten und Voice-Agents, besonders wenn Instruction-basierte Emotionssteuerung gewünscht ist.

---

3. Chatterbox – Der ElevenLabs-Herausforderer

Chatterbox von Resemble AI war einer der ersten Open-Source-Releases, der es 2025 schaffte, in Blindtests gegen ElevenLabs zu bestehen – und ist mittlerweile in Version **Multilingual V3** angekommen.

Highlights:

- 0,5B-Parameter-Modell mit **Llama-3-Backbone**
- Als erstes Open-Source-TTS-Modell mit **Emotion-Exaggeration-Control** (regelbare emotionale Übertreibung via `exaggeration`- und `cfg`-Parameter)
- Multilingual V3 deckt **23+ Sprachen** ab, inklusive dedizierter "Single Language Packs" für Chinesisch, Hindi und mehrere Spanisch-/Portugiesisch-Varianten
- Trainiert auf 0,5 Mio. Stunden Audiodaten
- Eingebautes **PerTh-Wasserzeichen** (überlebt MP3-Kompression) für Responsible-AI-Zwecke
- MIT-Lizenz – die freieste Lizenz unter allen hier verglichenen Modellen

- Über 2,5 Mio. Downloads/Monat auf Hugging Face

Einsatzgebiete: Sehr beliebt für kreative Inhalte (Memes, Videos, Games), Voice-Agents mit emotionalem Ausdruck – Referenz-Community-Favorit.

---

4. Kokoro – Klein, aber oho

Kokoro-82M ist das Gegenstück zu den riesigen Modellen: Mit nur **82 Millionen Parametern** liefert es laut Entwickler hexgrad Qualität, die mit deutlich größeren Modellen konkurrieren kann.

Highlights:

- Basiert architektonisch auf **StyleTTS 2**
- Extrem ressourcenschonend – läuft flüssig auf CPU, sogar im Browser (Kokoro.js via ONNX)
- Nutzt die eigene G2P-Bibliothek **misaki**
- Unterstützt mehrere Sprachvarianten (US-/UK-Englisch, Spanisch, Französisch, Hindi, Italienisch, Japanisch, Brasilianisches Portugiesisch, Mandarin)
- Apache-2.0-Lizenz, 8.100+ GitHub-Stars
- Kein Voice-Cloning-Feature – arbeitet mit vordefinierten Stimm-Presets (z. B. `af_heart`)

Einsatzgebiete: Perfekt für kosteneffiziente Massenproduktion von Sprache (Podcasts, Hörbuch-Vertonung, eingebettete Anwendungen), wo Server-Kosten und Latenz wichtiger sind als Voice-Cloning.

---

5. NeuTTS – On-Device-Champion

Die **NeuTTS**-Familie von Neuphonic (Air, Nano-Multilingual-Collection, 2E) ist explizit für den **offline, on-device Betrieb** konzipiert – bis hin zum Raspberry Pi.

Highlights:

- Verschiedene Backbone-Größen: NeuTTS-Air (~360M aktiv), NeuTTS-Nano (~120M aktiv), NeuTTS-2E (~125M aktiv, mit Emotionssteuerung)
- Eigener Audio-Codec **NeuCodec** (50Hz, Single-Codebook, hohe Qualität bei niedriger Bitrate)
- **GGUF-Quantisierungen** (Q4/Q8) für maximale Effizienz auf CPU
- Benchmark-Werte beeindruckend: Auf einem AMD Ryzen 9HX 370 erreicht NeuTTS-Nano 221 Tokens/Sekunde rein auf CPU
- Instant Voice Cloning mit nur 3 Sekunden Referenzaudio

- Unterstützt Englisch, Spanisch, Deutsch, Französisch (modellabhängig)
- Lizenzmix: NeuTTS-Air unter Apache 2.0, neuere Modelle (Nano, 2E) unter der "NeuTTS Open License 1.0"
- Eingebautes Perth-Wasserzeichen
- Auf GitHub bereits 6.200+ Stars, sehr aktive Entwicklung (fast täglich neue Commits)

Einsatzgebiete: Ideal für Privacy-first-Anwendungen, eingebettete Sprachassistenten, Spielzeuge oder Compliance-sensible Apps, bei denen keine Daten das Gerät verlassen dürfen.

---

6. VoxCPM2 – Tokenizer-freie Diffusion aus China

VoxCPM2 von OpenBMB (bekannt von den MiniCPM-Sprachmodellen) verfolgt einen architektonisch besonders interessanten Ansatz: Es ist **tokenizer-frei** und generiert kontinuierliche Sprachrepräsentationen direkt über ein End-to-End-Diffusion-Autoregressive-Modell.

Highlights:

- **2 Milliarden Parameter** – das größte Modell in diesem Vergleich
- Kein diskreter Audio-Tokenizer nötig – arbeitet direkt mit kontinuierlichen Repräsentationen, was Informationsverlust reduziert
- Unterstützt **30+ Sprachen**
- Ausgabe in **48kHz Studio-Qualität**
- Trainiert auf über 2 Millionen Stunden Sprachdaten
- Laut Community-Berichten erreicht VoxCPM2 im Similarity-Benchmark rund **85,4 %** bei englischer Sprache und gilt als eines der Modelle, die ElevenLabs bei der Sprecherähnlichkeit übertreffen
- State-of-the-art oder konkurrenzfähige Ergebnisse bei wichtigen Zero-Shot- und Controllable-TTS-Benchmarks
- Open Source über GitHub/Hugging Face

Einsatzgebiete: Für Anwendungen, die höchste Audioqualität (48kHz) und kontextbewusste, ausdrucksstarke Sprache brauchen – z. B. professionelle Synchronisation, Hörbücher, kreative Content-Produktion.

---

7. Gepard 1.0 – Der Sprint-Champion für Echtzeit-Dialoge

Gepard 1.0 (passenderweise nach dem schnellsten Landtier benannt) von **nineninesix** ist das jüngste und auf reine **Streaming-Performance** getrimmte Modell in diesem Vergleich.

Highlights:

- 555 Millionen Parameter, **Qwen3.5-Backbone**
- **Streaming-first**: Das Modell beginnt zu sprechen, sobald der erste Text-Chunk eintrifft – Audio wird frame-weise generiert
- Beeindruckende Performance-Kennzahlen: **~50ms Time-to-First-Audio (TTFA)** und **~20× Real-Time-Factor** auf einer einzelnen RTX 5090
- Explizit für **Real-Time-Dialogue-Synthese** konzipiert – also für Sprachassistenten und Konversations-KI, bei denen jede Millisekunde Latenz zählt
- Bereits Tag-0-Integrationsarbeit in vLLM-Omni in Vorbereitung
- Sehr frisch: erst vor wenigen Wochen open-sourced, aber schon große Aufmerksamkeit auf Reddit/HackerNews/X

Einsatzgebiete: Die klare Nummer 1 für latenzkritische Voice-Agent- und Conversational-AI-Anwendungen (z. B. Telefonie-Bots, Live-Übersetzung), wo eine Reaktionszeit im Millisekundenbereich entscheidend ist.

Direkter Vergleich nach Anwendungsfall

Anwendungsfall	Empfehlung
Maximale Sprachabdeckung (Nischensprachen)	**OmniVoice** (600+ Sprachen)
Bester Allround-Multilingual-Agent mit Instruction-Control	**Qwen3-TTS**
Emotionale/kreative Inhalte, MIT-Lizenz gewünscht	**Chatterbox**
Extrem günstige Massenproduktion (CPU/Browser)	**Kokoro**
Offline/On-Device/Privacy (Raspberry Pi, Mobile)	**NeuTTS**
Höchste Audioqualität (48kHz Studio-Sound)	**VoxCPM2**
Echtzeit-Dialog mit minimaler Latenz	**Gepard 1.0**

Gemeinsame Trends

Bei der Recherche fallen mehrere Muster auf, die die gesamte Open-Source-TTS-Szene 2026 prägen:

1. **LLM-Backbones sind Standard geworden**: Chatterbox nutzt Llama 3, NeuTTS und Gepard bauen auf Qwen-Backbones – TTS-Modelle werden zunehmend wie kleine Sprachmodelle behandelt.
2. **Voice Cloning mit wenigen Sekunden Audio** ist mittlerweile Basisfunktion, nicht mehr das Alleinstellungsmerkmal.
3. **Watermarking** (meist via Resemble AIs Perth) setzt sich als Responsible-AI-Standard durch – gleich mehrere Modelle (Chatterbox, NeuTTS) integrieren es standardmäßig.
4. **Streaming-Fähigkeit** wird zum entscheidenden Wettbewerbsfaktor – sowohl Qwen3-TTS als auch Gepard 1.0 werben explizit mit Latenzen unter 100ms.
5. **Apache 2.0/MIT dominieren** als Lizenzmodell, was kommerzielle Nutzung stark vereinfacht (Ausnahme: NeuTTS-Nano/2E mit eigener "Open License").

Fazit

Es gibt kein "bestes" TTS-Modell – die Wahl hängt vollständig vom Anwendungsfall ab. Wer **Reichweite** über möglichst viele Sprachen sucht, landet bei OmniVoice. Wer ein **produktionsreifes Allround-System** mit starker Instruction-Steuerung will, ist mit Qwen3-TTS gut bedient. Für **kreative, emotionale Sprachausgabe** unter freizügiger Lizenz bleibt Chatterbox der Community-Favorit. Wer **Kosten minimieren** will, greift zu Kokoro. **Datenschutz und Offline-Betrieb** sprechen für NeuTTS, **Studio-Qualität** für VoxCPM2 – und wer eine **blitzschnelle Sprachausgabe für Echtzeit-Konversationen** braucht, sollte sich Gepard 1.0 genau ansehen.

Die gute Nachricht: Alle sieben Modelle sind Open Source und lassen sich kostenlos selbst hosten – ein enormer Fortschritt gegenüber der Zeit, in der hochwertige TTS-Qualität ausschließlich hinter kostenpflichtigen APIs verschlossen war.