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. □□

Revision #1
Created 2026-07-04 23:30:10 UTC by art10m
Updated 2026-07-04 23:30:36 UTC by art10m