Roo Code: Anforderungen an LLMs und geeignete Open-Source-Modelle

Anforderungen von Roo Code an LLMs

- **Unterstützung von Tool-Use / Funktion Calls:** Das LLM muss in der Lage sein, externe **Tools oder Funktionen gezielt aufzurufen**, anstatt nur freien Text zu generieren. Roo Code vermittelt dem Modell eine Reihe von verfügbaren **Befehlen** (z. B. `read_file`, `write_to_file`, `execute_command` etc.), die das LLM bei Bedarf nutzen soll[^1]. Beispielsweise soll das Modell zum Anlegen einer Datei keinen Codeblock im Chat liefern, sondern einen strukturierten Tool-Aufruf wie `<write_to_file>…</write_to_file>` ausgeben[^2]. Diese Fähigkeit ähnelt dem *Function Calling* bei OpenAI-Modellen und erfordert, dass das LLM solche Aufrufe erkennt und korrekt formatiert ausgibt.
- **Strukturierte Ausgabe (JSON/Markup) für Tools:** Eng verbunden mit obigem Punkt ist die **Ausgabe in einem präzisen strukturierten Format**. Roo Code erwartet z. B. JSON- oder XML-artige Strukturen, damit die Erweiterung die Ausgaben maschinell parsen und ausführen kann[^3]. **Keine freien Erläuterungen oder Abweichungen** dürfen in diesen Abschnitten erscheinen, da sonst die Tool-Ausführung fehlschlagen könnte. Das LLM muss also Prompt-Anweisungen wie „Antworte **nur** mit einem gültigen JSON-Objekt" strikt befolgen können.
- **Dynamische Erweiterbarkeit via MCP:** Über die **Model Context Protocol (MCP)**-Schnittstelle können zur Laufzeit neue Tools/Funktionen hinzugefügt werden[^4][^5]. Das LLM sollte daher flexibel sein, **neue Tool-Definitionen im Prompt zu verstehen** und diese auf Nutzerwunsch einzusetzen. Beispielsweise kann ein Nutzer ein neues Tool wünschen („füge ein Tool hinzu, das X tut"); Roo Code stellt dem Modell dann die Beschreibung dieses Tools bereit, woraufhin das LLM dieses per `use_mcp_tool` oder `access_mcp_resource` aufrufen soll[^6][^7]. Die Anforderung an das Modell ist, dass es auch **ad-hoc definierte Funktionen** korrekt interpretieren und nutzen kann.
- **Kompatibilität mit Chat/API-Schnittstellen:** Roo Code kann sich mit **OpenAI-kompatiblen APIs** und benutzerdefinierten Modellschnittstellen verbinden[^8]. Das setzt voraus, dass das LLM im **Chat-Format** angesprochen werden kann (System-Prompt + Nutzer-Prompt + Assistenz-Antwort) und idealerweise eine API bietet, die **Funktionsaufrufe oder strukturierte Outputs** unterstützt. Falls das LLM keine native API bietet (z. B. lokale Modelle), muss es zumindest so trainiert oder instruiert sein, dass es im Chat-Kontext sinnvoll reagiert. **Rollen und Instruktionen** (System vs. User) müssen vom Modell verstanden und respektiert werden.
- **Hohe Prompt-Treue und kontrollierbares Verhalten:** Da Roo Code viel Logik in den System-Prompts unterbringt (z. B. detaillierte Anleitungen zur Tool-Verwendung oder wann *kein* Tool zu nutzen ist)[^9], muss das LLM diese Vorgaben **gewissenhaft befolgen**. Es sollte keine geheimen Systeminstruktionen „leaken" und soll sich an die vorgegebenen Modi halten (z. B. nur dann einen MCP-Server erstellen, wenn der Nutzer es verlangt[^10]). Kurz: Das Modell muss robust gegenüber Prompt Injection sein und strikt innerhalb der vorgesehenen Schnittstellen agieren. Nur dann kann Roo Code sicherstellen, dass die AI-Assistenten nicht vom vorgesehenen Ablauf abweichen.

Vergleich wichtiger Open-Source-LLMs hinsichtlich Kompatibilität

Modell	Tool-/Funktionsaufrufe	Strukturierte Ausgabe	Instruktions-Befolgung
Mistral 7B *(+ Codestral)*	✅ *Unterstützt Function-Calls* – Über die offizielle API von Mistral sind Funktionsaufrufe direkt möglich[^11]. Lokale Nutzung der Open-Weights erfordert zwar eigenes Prompting, aber das Modell wurde darauf ausgelegt, Tools nutzen zu können.	✅ *JSON-Ausgabe möglich* – Die Mistral-API bietet einen **JSON-Modus**, der strikt ein gültiges JSON-Objekt zurückliefert[^12].	✅ *Gute Befolgung* – Mistral-Modelle sind darauf ausgelegt, Anweisungen zu befolgen, und unterstützen strukturierte Ausgaben gut.

Modell	Tool-/Funktionsaufrufe	Strukturierte Ausgabe	Instruktions-Befolgung
LLaMA 2 Chat *(z. B. 13B/70B)*	⚠ *Kein nativer Function-Call*, aber über Prompt machbar – Meta's LLaMA-Modelle bieten von Haus aus keine API für Funktionsaufrufe. Entwickler können jedoch dank Open-Source-Zugriff eigene Anpassungen vornehmen[^13]. In Roo Code lässt sich LLaMA 2 via OpenAI-kompatible Schnittstellen (z. B.**LM Studio/Ollama**als lokaler Server) einbinden und durch Few-Shot-Beispiele dazu bringen, Tools per Markup aufzurufen.	⚠ *Formatierung bedingt zuverlässig*– LLaMA 2 Chat wurde auf Befolgung von Formatwünschen trainiert, kann also**JSON oder XML**ausgeben, neigt aber manchmal dazu, zusätzlich erklärenden Text zu produzieren. Größere Varianten (70B) sind hierbei verlässlicher als kleinere. Mit strengem Prompt (z. B. „Antworte**nur**mit JSON") erzielt man meist korrekte strukturierte Outputs.	✅ *Sehr hohe Prompt-Treue*– Die RLHF-Chatvarianten von LLaMA 2 respektieren System-Prompts und Rollen in der Regel genau. Unerlaubte Inhalte werden vermieden und Anweisungen (etwa nur bestimmte Tools zu verwenden) befolgt das Modell zuverlässig.**Neue Tool-Definitionen**im Kontext werden verstanden, aber das Modell hat keine spezielle Ausbildung darin, sodass es evtl. weniger**initiative**zeigt, Tools autonom vorzuschlagen.
OpenChat 13B *("GPT4All-J" Variante)*	⚠ *Kein spezieller Tool-Modus*– OpenChat ist eine RLHF-verfeinerte LLaMA-13B-Version, die primär auf Dialogqualität optimiert ist[^14]. Es besitzt keine eingebauten Function-Call-APIs. Tools kann es dennoch nutzen, wenn im Systemprompt entsprechend angeleitet (ähnlich wie bei LLaMA 2). Eigenständig Funktionen aufzurufen (wie OpenAI GPT-4) wurde nicht explizit trainiert.	⚠ *Strukturierte Antworten mit Nachhilfe*– Da OpenChat auf qualitativ hochwertige Gespräche getrimmt wurde, kann es auf Aufforderung JSON/Code-Blöcke ausgeben. Eine Garantie für fehlerfreie**JSON-Tool-Aufrufe**gibt es aber nicht. In einfachen Fällen hält es das Format ein, bei komplexen verschachtelten Strukturen könnte Nachbearbeitung nötig sein.	✅ *Folgsam und dialogstark*– Durch das RLHF-Tuning auf GPT-4-Daten ist OpenChat sehr**hilfsbereit und präzise**in der Befolgung von Anweisungen. Systeminstruktionen werden ernst genommen. Im Vergleich zu weniger feinjustierten Modellen ist OpenChat weniger widerspenstig und dürfte neue Tools im Prompt kontextgerecht einsetzen, sobald der Nutzer dies wünscht.
Nous-Hermes 13B *(+ Hermes 2/3)*	✅ *Gezielt auf Tool-Use trainiert*– Die Hermes-Reihe (auf Basis LLaMA 2) von Nous befähigt LLMs **explizit zum Funktionsaufruf**. Hermes 2 und 3 wurden mit speziellen System-Prompts und Beispielen für Tool-Nutzung fine-getuned[^15]. Das Modell kann definierte Funktionen als JSON-Aufruf ausgeben (inkl. Funktionsname und Argumenten) statt freiem Text[^16]. Diese Fähigkeit ist nativ (kein externer Parser nötig) – das Inferenz-Skript erkennt `{{ ... }}`Blöcke und führt sie aus.	✅ *Sehr zuverlässige Struktur*– Dank Feintuning auf *Function-Calling*-Schemas hält Hermes sich strikt an das vorgegebene Format (z. B. Pydantic-Schema für Tool-Argumente)[^17]. In Tests liefert Hermes die**gewünschten JSON-Objekte**praktisch ohne Ausschweifungen oder Fehler, sofern der Systemprompt korrekt gesetzt ist. Dieses modellinterne Protokoll reduziert das Risiko von Formatfehlern deutlich.	⚠ *Hohe Compliance, aber geringere Filterung*– Hermes ist darauf ausgelegt, **Benutzerwünsche maximal zu erfüllen**. System- und Nutzeranweisungen werden akkurat umgesetzt. Allerdings wurde bewusst etwas weniger auf „Alignment" (sicheres/harmloses Verhalten) geachtet[^18], um die Nützlichkeit zu steigern. Für Roo Code heißt das: hervorragende Tool-Befolgung, jedoch sollte man unbeabsichtigte Befehle (z. B. destruktive Aktionen) weiterhin durch Roo Code's Approval-Mechanismen absichern.

Modell	Tool-/Funktionsaufrufe	Strukturierte Ausgabe	Instruktions-Befolgung
Zephyr 7B (Beta) *von HuggingFace*	⚠ *Keine eingebauten Function-Calls*– Zephyr ist ein auf Mistral 7B basierender Chat-Assistent, primär optimiert auf hilfreiche Antworten[^19]. Es bietet **keine spezielle Tools-API**, lässt sich aber analog zu anderen 7B-Instruct-Modellen in Roo Code einbinden. Tools-Aufrufe müssen via Prompt vorgegeben werden. Zephyr selbst wird nicht „wissen", wann ein Tool angebracht ist, ohne entsprechende Systeminstruktion.	⚠ *Kann formatiert antworten*– Als hilfreicher Chatbot kann Zephyr durchaus JSON oder Code formatiert liefern, wenn man es verlangt. Aufgrund der nur 7 Mrd. Parameter und dem Verzicht auf strenge RLHF kann es aber vorkommen, dass das Modell Antworten leicht außerhalb der geforderten Form gibt. Bei klarer Vorgabe (etwa durch Beispiel-Ausgaben im Prompt) ist es in der Lage,**korrekt strukturierte**Tool-Kommandos zu erzeugen.	✅/⚠ **Befolgung vs. Freiheit** – Zephyr wurde mittels Direct Preference Optimization auf Nützlichkeit getrimmt, mit reduzierter „Über-Anpassung" an menschliche Feedbackdaten[^20]. Es **folgt Nutzeranweisungen sehr direkt**, was positiv für Tool-Nutzung ist. Gleichzeitig fehlt etwas die strikte Regelbefolgung eines RLHF-Modells – d. h. es gibt weniger innere Hemmungen, aber evtl. muss das Systemprompt es stärker lenken. Insgesamt kann Zephyr neue Instruktionen (wie zusätzliche MCP-Tools) verstehen, benötigt aber eine präzise Führung, damit es sich an das gewünschte Protokoll hält.

Empfehlung für geeignete LLMs

Zusammenfassend eignen sich **Open-Source-LLMs mit spezialisierter Tool-Use-Fähigkeit oder hoher Format-Treue** am besten für Roo Code:

- **Nous Hermes (Function-Call-tuned)** – Besonders die neueren Hermes-Versionen sind darauf ausgelegt, **JSON-basierte Toolaufrufe** zu generieren. Sie erfüllen Roo Codes Anforderungen praktisch am genauesten, da sie das benötigte Verhalten (Tools per Schema nutzen) schon „gelernt" haben[^21]. Für einen Entwickler bedeutet das weniger Prompt-Tuning und eine höhere Zuverlässigkeit bei der Automation.
- **Mistral (via offiz. API oder Codestral)** – Mistral 7B überzeugt durch **eingebaute Funktion-Calling-Unterstützung** und einen erzwingbaren JSON-Antwortmodus[^22][^23]. In Kombination mit dem Code-spezialisierten *Codestral*-Modell kann Mistral viele Entwicklungsaufgaben abdecken. Trotz der geringeren Modellgröße liefert es dank API-gestützter Strukturvorgaben robuste Ergebnisse und lässt sich nahtlos in Roo Code integrieren.
- **LLaMA 2 70B Chat (oder hochwertige Fine-Tunes wie OpenChat 13B)** – Falls maximale Code-Verständnis und Kontextlänge wichtig sind, bietet ein größeres LLaMA-Modell die **stärkste allgemeine Leistung**. Es hat zwar keine native Tool-API, folgt aber Anleitungen sehr genau. Mit sorgfältigem Systemprompt (inkl. Tool-Beschreibungen und Beispielen) kann LLaMA 2 70B zuverlässig in Roo Codes Rolle als „AI-Entwickler" agieren und komplexe Aufgaben bearbeiten. OpenChat 13B ist eine leichtere Alternative, die bereits mit hohem Gesprächsniveau punktet und nur minimalen Feinschliff für die Tool-Nutzung erfordert.

Im Ergebnis sollte die Wahl des Modells die **Balance aus Fähigkeit zur strukturierten Tool-Ausgabe und allgemeiner Modellqualität** widerspiegeln. Modelle wie Hermes und Mistral punkten bei der direkten Tool-Einbindung, während größere LLaMA-Derivate durch bessere Sprachverständnis- und Planungsfähigkeiten glänzen. Für Roo Code sind daher insbesondere jene LLMs empfehlenswert, die **out-of-the-box JSON/Function-Calls unterstützen** oder sich mit geringem Aufwand dahingehend konditionieren lassen, ohne die Zuverlässigkeit im Dialog zu verlieren.

Referenzen

[^1]: How Tools Work | Roo Code Docs
[^2]: How Tools Work | Roo Code Docs
[^3]: How Tools Work | Roo Code Docs
[^4]: GitHub - RooVetGit/Roo-Code
[^5]: Move MCP creation from system prompt to agent/tool or MCP server · RooVetGit Roo-Code · Discussion #558 · GitHub
[^6]: Move MCP creation from system prompt to agent/tool or MCP server · RooVetGit Roo-Code · Discussion #558 · GitHub
[^7]: Move MCP creation from system prompt to agent/tool or MCP server · RooVetGit Roo-Code · Discussion #558 · GitHub
[^8]: GitHub - RooVetGit/Roo-Code
[^9]: Move MCP creation from system prompt to agent/tool or MCP server · RooVetGit Roo-Code · Discussion #558 · GitHub
[^10]: Move MCP creation from system prompt to agent/tool or MCP server · RooVetGit Roo-Code · Discussion #558 · GitHub
[^11]: Using Mistral AI With Roo Code | Roo Code Docs
[^12]: Using Mistral AI With Roo Code | Roo Code Docs
[^13]: Top 6 LLMs that Support Function Calling for AI Agents
[^14]: openchat/openchat · Hugging Face
[^15]: GitHub - NousResearch/Hermes-Function-Calling

[^16]: GitHub - NousResearch/Hermes-Function-Calling
[^17]: GitHub - NousResearch/Hermes-Function-Calling
[^18]: HuggingFaceH4/zephyr-7b-beta · Hugging Face
[^19]: HuggingFaceH4/zephyr-7b-beta · Hugging Face
[^20]: HuggingFaceH4/zephyr-7b-beta · Hugging Face
[^21]: GitHub - NousResearch/Hermes-Function-Calling
[^22]: Using Mistral AI With Roo Code | Roo Code Docs
[^23]: JSON mode | Mistral AI Large Language Models