ChatGPT

Agentische Workflows und Vibe-Coding: Stand der Technik

1. Technische Frameworks und Architekturen für agentische Workflows

LangGraph (LangChain): *LangGraph* ist ein ereignisgetriebenes Orchestrierungs-Framework zum Aufbau agentischer Systeme [1]. Es bietet sowohl **deklarative** (Graph-basiert) als auch **imperative** APIs, mit denen sich Workflows als **Graph aus Knoten und Kanten** modellieren lassen [2]. Jeder Knoten repräsentiert eine Funktion oder Aufgabe, und Kanten definieren Übergänge – diese können fest oder konditional sein, sodass der Ablauf je nach Ergebnis verzweigen kann [3]. LangGraph zeichnet sich durch hohe **Modularität** aus: Die innere Logik jedes Knotens wird in normalem Code (Python/TypeScript) implementiert, während die Graph-Struktur den Ablauf steuert [4]. Besonderen Wert legt LangGraph auf **Persistenz** und **Fehlertoleranz**: Ein eingebauter Persistenz-Layer ermöglicht es, den Zustand zu speichern, was **Unterbrechbarkeit**, **Fehlerbehandlung** (Wiederaufnahmen) und **Memory** (Kurz- und Langzeit) erleichtert [5]. So können sogar Human-in-the-Loop-Interaktionen (z. B. *Interrupt*, *Approve*, *Resume*) realisiert werden [6]. LangGraph unterstützt **Tool-Aufrufe** (z. B. Funktionen binden) und **strukturierte Outputs** out-of-the-box [7] [8]. Insgesamt kombiniert LangGraph starre Workflows mit dynamischen Agenten: Entwickler können *Workflows* für vorgegebene Abläufe definieren **und** *Agents* für flexible, vom Modell gesteuerte Schritte verwenden [9]. Diese Mischung ist praxisnah, da produktive agentische Systeme oft deterministische Sequenzen mit LLM-Entscheidungen kombinieren [9]. Durch Integration mit LangChain-Tooling (LangSmith) bietet LangGraph zudem **Debugging**, **Streaming** (Token-Streaming, Ereignis-Streaming) und **Deployment**-Unterstützung [10].

CrewAI: *CrewAI* ist eine Multi-Agenten-Plattform, die speziell auf Unternehmen abzielt. Sie ermöglicht es, **KI-Workflows** zu orchestrieren, bei denen mehrere spezialisierte Agenten (als *Crew* oder Team) zusammenarbeiten [11]. CrewAI kombiniert zwei Paradigmen: **Flows** und **Crews** [12]. Ein **Flow** ist ein strukturierter, ereignisgetriebener Ablauf – ähnlich einem Workflow – mit bedingter Logik, Schleifen und Zustandsverwaltung [13]. Dies erlaubt deterministische Mehrschritt-Automationen (vergleichbar mit LangGraph-Workflows). **Crews** hingegen sind dynamische Teams aus Agenten, wobei jeder Agent eine **Rolle**, **Ziel(e)**, eine eigene **Backstory** und bestimmte **Tools** hat [14] [15]. Durch diese rollenbasierte Aufteilung kann CrewAI komplexe Aufgaben durch **autonome Arbeitsteilung** bewältigen: Die Agenten kommunizieren untereinander, delegieren Unteraufgaben und kombinieren ihr Wissen, um Probleme zu lösen [16] [17]. CrewAI fördert **Modularität** durch klar getrennte Komponenten (Agenten, Aufgaben, Tools, Prozesssteuerung) [18] [19]. **Tool-Integration** ist fester Bestandteil – Tools (z. B. Datenbankzugriff, API-Aufrufe) werden einzelnen Agenten als Fähigkeiten zugewiesen [19]. Ein **Process Layer** koordiniert die Interaktion der Agenten, synchronisiert Aufgaben und verwaltet die Kommunikation, um reibungslose Abläufe sicherzustellen [20]. In Bezug auf **Fehlerhandling** und Robustheit bietet CrewAI vor allem Monitoring- und Logging-Funktionen auf Enterprise-Niveau: Es gibt Dashboards für detaillierte Logs, Performance-Metriken und Systemgesundheit, sodass menschliche Operatoren Engpässe erkennen und eingreifen können [21] [22]. CrewAI eignet sich insbesondere, wenn man **planbare Abläufe** (über *Flows*) mit der **Flexibilität von Agenten** kombinieren möchte. Beispielsweise kann ein Flow eine Pipeline definieren, in der an bestimmten Schritten ganze Crews von Agenten eingeschaltet werden, um Teilprobleme zu bearbeiten [23] [24]. Diese klare Trennung zwischen deterministischen Abläufen und Agenten-Kollaboration ist ein zentrales Design-Merkmal.

AutoGen (Microsoft): *AutoGen* ist ein Open-Source-Framework von Microsoft Research zur Entwicklung agentischer KI-Anwendungen [25]. Im Kern ermöglicht AutoGen die Erstellung von Anwendungen mit **mehreren LLM-Agenten**, die miteinander kommunizieren, um Aufgaben zu lösen [26]. Bekannt wurde AutoGen durch das *"Commander & Writer"*-Paradigma, bei dem etwa ein *Planer*-Agent (Commander) Aufgaben zerlegt und Anweisungen gibt, ein *Ausführender* Agent (Writer) Lösungen vorschlägt, und ggf. ein *Safeguard*-Agent die Ergebnisse überprüft [27]. Die neueste Version AutoGen v0.4 (Ende 2024) setzt stark auf eine **asynchrone, event-gesteuerte Architektur** für bessere Skalierbarkeit [28]. **Modularität** wird großgeschrieben: Entwickler können eigene Agententypen, Tools, Speichermodule und Modell-Anbindungen als **Plug-ins** integrieren [29]. AutoGen erlaubt sowohl klassische Anfrage-Antwort-Muster als auch **ereignisgetriebene Interaktionen** zwischen Agenten (Agents können proaktiv Nachrichten austauschen) [30]. Zudem unterstützt es **langlaufende** Agenten, die im Hintergrund weiterarbeiten. In Sachen **Planungsfähigkeit** bietet AutoGen keine starre Workflow-Notation wie LangGraph, dafür aber flexible **Mehragenten-Konversationen**: Entwickler definieren, welche Agenten mit welchen Rollen (z. B. *User-Simulator*, *Assistent*, *Reviewer*) an einer Konversation teilnehmen und wie sie aufeinander reagieren. Komplexe Kollaborationsmuster (bspw. iterative Verbesserungszyklen) lassen sich so in natürlicher Sprache oder in Code definieren [26]. **Tool-Integration** erfolgt über ein *Extensions*-Modul – z. B. gibt es Erweiterungen für Code-Ausführung in Docker, für die Anbindung an OpenAI-Modelle oder gRPC-Kommunikation [31] [32]. AutoGen legt ferner großen Wert auf **Beobachtbarkeit und Debugging**: eingebautes **Tracing**, Logging und Unterstützung für OpenTelemetry erlauben es, die Interaktionen der Agenten nachzuvollziehen [33] [34]. Insgesamt zielt AutoGen darauf ab, **deterministische Abläufe** mit **dynamischer Agenten-KI** zu verbinden – beispielsweise für unternehmensweite Prozessautomatisierung [35]. Fehlerhandling wird vor allem durch Architektur sichergestellt: Asynchrone Events verhindern Blockierungen, und Entwickler können auf **Events** (z. B. Fehler, Zeitüberschreitungen) programmatisch reagieren. Durch Community-Plugins und plattformübergreifende Unterstützung (Python, .NET) ist AutoGen zu einem vielseitigen Baukasten für agentische Workflows geworden [36] [37].

LangChain Agents: Die *LangChain*-Bibliothek bietet seit 2022 *Agents* als Abstraktion, um LLMs mit Werkzeugen interagieren zu lassen. Ein LangChain-Agent besteht typischerweise aus einem LLM (z. B. GPT-4) plus einer Menge definierter **Tools** (Funktionen, API-Zugriffe, Wissensdatenbanken etc.) [38]. Das System verwendet *Prompt-Vorlagen*, welche das Modell anleiten, in einem **Schleifenmodus** zu arbeiten [39]. Konkret läuft ein LangChain-Agent so ab: Das LLM erhält den Nutzerauftrag plus einen *Scratchpad* (eine Zwischennotizfläche) und entscheidet Schritt für Schritt, ob es eine Aktion (Tool-Aufruf) durchführen will. Wenn ja, gibt es einen bestimmten *Aktions-String* aus (z. B. `Search["Frage"]`), woraufhin LangChain den entsprechenden Tool-Call ausführt. Die Ergebnis-Observation (z. B. Suchtreffer) wird ins Prompt zurückgeführt, und das LLM setzt seine **Gedankenkette** fort [40]. Dieses Looping (bekannt als ReAct-Muster) wiederholt sich, bis das Modell zum Schluss kommt und eine Antwort generiert [39]. Die **Tool-Integration** ist bei LangChain sehr ausgereift: Es existieren zahlreiche vordefinierte Tools (Datenbanksuche, Web-Suche, Rechner, …) und eine einfache API, um eigene Tools zu definieren. Beschreibungen der Tools werden ins Prompt eingebettet, damit das LLM weiß, wie und wann es sie nutzen kann [39] [41]. **Modularität:** LangChain stellt verschiedene *Agent-Typen* bereit (z. B. **Zero-shot ReAct**, **Conversational Agent**, **Plan-and-Execute**), die unterschiedliche Prompt-Strategien verwenden. Entwickler können relativ einfach zwischen diesen wählen oder eigene Agent-Vorlagen erstellen. Ein LangChain-Agent ist nicht so formal graphisch modellierbar wie bei LangGraph, dafür schnell prototypisierbar. **Planungsfähigkeit:** Ursprünglich waren LangChain-Agents rein **reaktiv** (der Plan entsteht on-the-fly durch Chain-of-Thought). Neuere Ansätze in LangChain erlauben auch zweistufige Verfahren – z. B. erst einen Plan entwerfen, dann ausführen (ähnlich *BabyAGI*). Dennoch liegt der Fokus auf schnellem *Aktionsentscheiden* pro Iteration, statt langfristiger Planung. **Fehlerbehandlung:** LangChain kümmert sich um Abbruchkriterien (etwa maximale Schleifenanzahl, Fallback wenn kein Tool passt) und formatiert Fehlermeldungen von Tools zurück ans LLM. Allerdings muss der Entwickler selbst dafür sorgen, dass der Agent bei falschem Verhalten korrigiert wird – z. B.

durch zusätzliche Regeln im Prompt (“Wenn das Ergebnis unsinnig ist, versuche einen anderen Ansatz”). In Kombination mit LangChain’s Memory-Modulen können Agents mehrturnfähig gemacht werden (sie behalten vergangene User-Eingaben und Agent-Aktionen im Kontext). Insgesamt ist LangChain ein **praktisches Framework**, um Agenten schnell zu implementieren, mit starker Integration von Tools, aber es erfordert häufig Fein-Tuning der Prompts, um die **Zuverlässigkeit** in komplexen Fällen sicherzustellen [42] [43]. (Die Herausforderungen – z. B. Kontext vollständig halten, Halluzinationen vermeiden – gelten hier wie generell bei Agents.)

TaskWeaver: *TaskWeaver* (2024 von Microsoft open-sourced) verfolgt einen **Code-zentrierten Ansatz** für LLM-Agenten [44]. Anstatt dass das LLM direkt natürliche Sprache ausgibt oder Tool-Strings konstruiert, übersetzt TaskWeaver Benutzeranfragen in **ausführbaren Code** [45]. Die Idee dahinter: Viele Probleme (insb. in Datenanalyse oder Automatisierung) lassen sich lösen, indem man geeigneten Code generiert und ausführt. TaskWeaver stellt dem Agenten daher eine Laufzeitumgebung bereit, in der er Python-Code schreiben und ausführen kann. **Plugins** (vom Nutzer definierte Funktionen/Bibliotheken) werden dem Agenten als *callable* Funktionen angeboten – also analog zu Tools [44]. Wenn der Agent also eine Datenbank abfragen soll, würde er in TaskWeaver z. B. Code mit einem Aufruf `query_db("Frage")` generieren; diese Funktion ist als Plugin definiert und führt die Aktion tatsächlich aus [44]. Dadurch kann der Agent auch mit **reichhaltigen Datenstrukturen** umgehen (Listen, Tabellen, Objekte), weil er mittels Code komplexe Logik ausführen kann, anstatt alles über Strings zu machen. **Modularität:** Der Entwickler kann beliebige Plugins/Funktionsbibliotheken bereitstellen – TaskWeaver behandelt sie als Erweiterung des *Wortschatzes* des LLM. **Planung:** Interessanterweise übernimmt das LLM hier implizit die Planung, indem es mehrschrittigen Code schreibt. Es kann z. B. erst Zwischenergebnisse berechnen, diese in Variablen speichern und dann weiterverwenden – all das mittels generierten Codes. Der Agent hat also eine Art *Scratchpad* in Form eines Python-Skripts. Das verleiht ihm gewissermaßen **planerische Fähigkeiten**, da er Teilschritte im Code festhalten kann, anstatt sie nur im flüchtigen Denken zu haben. **Tool-Integration:** Ist nahtlos – jedes Plugin ist letztlich eine Tool-Funktion. TaskWeaver unterstützt dynamische **Plugin-Auswahl**: Das LLM entscheidet selbst, welche Plugins in welcher Reihenfolge aufgerufen werden, indem es entsprechenden Code schreibt [44]. **Fehlerbehandlung:** Wenn der generierte Code Fehler wirft (Exceptions), kann der Agent diese sehen und daraus lernen (z. B. den Code korrigieren). TaskWeaver kümmert sich um **Sicherheit** – der Code wird abgeschottet (Sandbox) ausgeführt, um Schaden zu vermeiden [46]. Insgesamt ist TaskWeaver besonders nützlich in Szenarien, wo die Aufgabe viel strukturiertes **Programmieren** erfordert (z. B. Datenanalysen, die Abfragen und Berechnungen umfassen) [47]. Tests zeigen, dass dieser Ansatz domain-spezifische Aufgaben oft flexibler löst als rein sprachbasierte Agents [44]. Allerdings braucht das Modell ausreichende Code-Kompetenz, was bei modernen Codex-ähnlichen Modellen gegeben ist.

Zum Vergleich der genannten Frameworks zeigt **Tabelle 1** die Schwerpunkte:

Framework	**Modularität** (Architektur)	**Planungs-** **fähigkeit**	**Tool-Integration**	**Fehler-** **handling**
LangGraph	Nodes/Edges-Graph, persistent (Memory, Human-in-Loop) [5]. Sehr modulare Orchestrierung via deklarativem Graph und imperativem Code pro Node.	Kombination von Workflows (deterministisch) und Agenten (dynamisch). Conditional Branching ermöglicht flexible Pläne [4].	Tools als first-class citizens (LLM mit Werkzeugen augmentiert) [7] [8]. Integration mit LangChain-Tooling (viele Schnittstellen).	Persistenz erlaub Wiederaufnahme nach Fehlern; Fault Tolerance eingebaut (Logs, Zeitreisen) [5]. Debugging/ Monitoring via LangSmith.

Framework	**Modularität** (Architektur)	**Planungs-** **fähigkeit**	**Tool-Integration**	**Fehler-** **handling**
CrewAI	Modularer Multi-Agent-Ansatz: Agenten mit Rollen, Aufgaben, Tools; Flows als strukturierte Pipelines [13] [14] .	Zweistufig: vordefinierte *Flows* für Routineabläufe; *Crews* (Teams) für adaptive Problemlösung. Delegation und Inter-Agent-Kommunikation als “Plan” im Team [16] .	Tools modulhaft pro Agent zugewiesen (APIs, DB etc.) [19] . Integration externer KI-Dienste (z.B. via Bedrock) unterstützt [48] [49] .	Enterprise-Level Monitoring (Logging-Dashboards) [21] ; Möglichkeit zur menschlichen Überwachung/ Eskalation. Fehler abfangen durch Prozess-Controlle
AutoGen	Asynchrone, ereignisgesteuerte Architektur [28] . Hochgradig erweiterbar durch Plugins/ Extensions [29] ; Multi-Agent mit eigenen Kommunikationsprotokollen.	Kein fixer Ablaufplan, dafür flexible Agenten-Konversation. Rollenverteilung (Commander/Worker) ermöglicht Aufgabenteilung (Planen/Ausführen). Agenten können proaktiv agieren.	Tools über Extensions (z.B. Code-Execution, Internetzugriff) integrierbar [32] . Community-Plugins für viele Anwendungsfälle verfügbar [50] . Cross-Language (Python, .NET) Unterstützung.	Robustes Event-System vermeide Deadlocks. Eingebaute Observability (Tracing, OpenTelemetry) [33] erleichtert Fehlersuche. Safeguard-Agenten können Ausgaben filtern (Wertausrichtung
LangChain Agent	Modular via Agent-Klassen (ReAct, Conversational, etc.) und Memory-Komponenten. Einfache Konfiguration durch Prompt-Templates.	Hauptsächlich reaktive Ein-Schritt-Planung (Gedanken + Aktion iterativ) [39] . *Plan-and-Execute*-Ansätze experimentell verfügbar, aber nicht Kern.	Sehr breite Tool-Palette out-of-the-box. Tools als Python-Funktionen mit Beschreibung; dynamischer Werkzeugaufruf in Schleife [39] .	Basis-Fehlerbehandlun (max. Schritte, falsche Tool-Outputs) in Framework. Für hohe Verlässlichkeit of zusätzl. Heuristiken im Prompt nötig. Memory kann genutzt werden, um Konsistenz über Turns zu wahren.

Framework	**Modularität** (Architektur)	**Planungs-** **fähigkeit**	**Tool-Integration**	**Fehler-** **handling**
TaskWeaver	Code-zentriert, Plugins als modulare Funktions-Bausteine [44]. LLM generiert Code, der verschiedene Module kombiniert.	LLM plant implizit via Codeaufbau: komplexe Logik durch Hilfsvariablen, Schleifen etc. möglich. Nicht linearer Plan (kann Zwischenresultate nutzen).	Plugins = Tools (z.B. DB-Query, Webscraper) als aufrufbare Funktionen [44]. LLM entscheidet, welche Funktion wann, indem es entsprechenden Code schreibt.	Fehler beim Code (Exceptions) werden sichtbar und können im nächsten Durchlauf berücksichtigt werden. Ausführung in Sandbox für Sicherheit [46]. Agent kann Code überarbeiten, fall initial fehlerhaft.

Tabelle 1: Vergleich ausgewählter Frameworks für agentische Workflows in Bezug auf Modularität, Planung, Tool-Einbindung und Fehlermanagement.

2. Theoretische Konzepte und konzeptionelle Ansätze für agentisches Reasoning

Moderne Agentenframeworks werden von einer Reihe theoretischer Konzepte untermauert. Diese Konzepte adressieren die **Entscheidungsfindung** eines Agenten, seine **Interaktion** mit der Umgebung (oder anderen Agenten) und Mechanismen zur **Selbstkorrektur**. Im Folgenden werden einige wichtige Ansätze vorgestellt:

- **ReAct (Reason + Act):** Der *ReAct*-Ansatz (Yao et al., 2022) kombiniert sprachliches **Reasoning** (Denken in natürlicher Sprache) mit **Action** (Tool-Nutzung) in einem einzigen ganzheitlichen Prozess [51]. Das Konzept sieht vor, dass ein LLM **interleaved** denkt und handelt: Es produziert abwechselnd *Gedanken* („Thought…“) und *Aktionen* („Action…“) sowie verarbeitet *Beobachtungen* dazu [51]. Diese Verschränkung erlaubt dem Modell, sein weiteres Vorgehen auf Basis der Ergebnisse von Tool-Aufrufen auszurichten. ReAct war insofern bahnbrechend, als es zeigte, dass ein LLM durch prompt-basiertes *Chain-of-Thought* in Kombination mit Aktionen komplexere Aufgaben lösen kann, als wenn es nur denkt oder nur handelt [52] [53]. Entscheidungsfindung erfolgt hier iterativ: Das Modell plant keinen vollständigen Lösungsweg im Voraus, sondern entscheidet Schritt für Schritt, basierend auf den neuesten Observations. ReAct bildet das Grundmuster vieler heutiger Agenten (etwa LangChain's Standard-Agent). Für das **agentische Reasoning** bedeutet ReAct, dass *kognitive Schritte* (Schlussfolgern, Unterfragen generieren) und *operationelle Schritte* (Tool-Einsatz) eng verzahnt sind – was zu besser informierten Entscheidungen führt. Beispiel: Ein Agent soll eine Frage beantworten, die Web-Suche erfordert. Mit ReAct kann er erst überlegen („Ich sollte Wikipedia durchsuchen nach X“), dann die Aktion `Search[X]` durchführen, das Resultat lesen und weiter schlussfolgern, bis die Antwort gefunden ist. Dadurch entsteht ein dynamischer Plan on-the-fly.

- **CAMEL (Communicative Agents for Multi-Agent Role-Play):** *CAMEL* (Li et al., 2023) bezeichnet ein Framework, in dem zwei oder mehr LLMs in **Rollenspielen** miteinander interagieren [54]. Die Idee: Anstatt dass ein Mensch beide Seiten eines Dialogs führt, übernimmt z. B. ein *KI-„Benutzer“-*

Agent die Rolle des Auftragsgebers und ein *KI-„Assistent"-Agent* die Rolle des Lösers. Mittels **Inception Prompting** – d. h. geschickten System-Prompts – wird festgelegt, welche Rollen, Persönlichkeiten und Ziele die Agenten haben [55]. Dann lässt man sie autonom miteinander chatten, um eine Aufgabe zu lösen. Dieses Setup fördert *agentisches Reasoning* auf zwei Ebenen: Zum einen muss jeder Agent **Entscheidungen** treffen, was er als Nächstes sagt oder tut (ähnlich dem Einzelagenten-Reasoning). Zum anderen ergibt sich aus der **Interaktion** ein kooperatives Problemlösen. CAMEL demonstrierte, dass LLM-Agenten über Hunderte von Dialog-Turns kooperativ Aufgaben bearbeiten können, **ohne menschliche Eingriffe**, solange die initialen Rollen gut gewählt sind [54]. Beispielsweise kann man einen *Programmierer-Agent* und einen *Manager-Agent* schaffen, die zusammen ein Coding-Problem angehen – der Manager formuliert Anforderungen, der Programmierer liefert Entwürfe, der Manager prüft usw. CAMELs Beitrag zum theoretischen Verständnis agentischer Systeme liegt darin, zu zeigen, dass **soziale Abläufe** (Fragen, Erklären, Vorschläge machen, Kritik üben) zwischen Agenten emergent entstehen können, wenn ihre Ziele geeignet definiert sind. Entscheidungsfindung erfolgt hier in Form von **Dialogakten**: Jeder Agent wählt seine nächste Äußerung basierend auf der bisherigen Konversation und seiner Rolle. CAMEL betont auch **Kooperation vs. Konkurrenz** als Designfrage – in den meisten Fällen ist Kooperation gewünscht (Agents arbeiten gemeinsam am selben Ziel). Dieser Ansatz liefert Erkenntnisse darüber, wie man Mehr-Agenten-Systeme stabil hält und aligniert (z. B. durch gemeinsame Ziele oder Moderationsmechanismen).

- **Toolformer:** *Toolformer* (Schick et al., 2023, Meta AI) ist weniger ein Laufzeit-Framework als ein **Trainingskonzept**, aber hochrelevant für agentische Fähigkeiten. Es zeigt, dass ein Sprachmodell lernen kann, **selbstständig** zu entscheiden, wann und wie externe Tools aufgerufen werden sollen [56]. Konkret wurde ein Language Model mit geringem überwachten Aufwand so feinjustiert, dass es in der normalen Textgeneration spezielle *API-Call-Token* einfügt (z. B. ein `<calc>`-Token für einen Taschenrechner) [56]. Das Modell erhielt einige Beispiele, wie man bestimmte Tools verwendet (etwa für Rechnen, Übersetzen, Suchen) – und danach konnte es **in neuer Umgebung** diese Tools einsetzen, um seine Antworten zu verbessern. Theoretisch demonstriert Toolformer, dass LLMs **intern planen** können: Während des Textgenerierens entscheiden sie, an welcher Stelle ein Werkzeug zu Hilfe genommen werden muss, und sie formatieren den Aufruf korrekt (Argumente, Syntax etc.) [56]. Für agentic reasoning bedeutet das, die **Werkzeugnutzung ist Teil der Sprache** des Modells. Anders formuliert: Das Modell erweitert seine *Action Space* um API-Aufrufe, ähnlich wie ReAct dies mit textualen Aktionen macht [41]. Toolformer verbessert die Fähigkeit zur **wissensbasierten Entscheidung** – es nutzt Tools vor allem, um Lücken zu füllen (z. B. Rechenaufgaben, Wissensfragen). Interessant ist, dass Toolformer im Training keine starken Aufsicht benötigte – es "überwacht" sich quasi selbst durch die Ergebnisse der Tools (selbst-supervised). Dieses Konzept beeinflusst neuere Agenten-Ansätze dahingehend, Tools noch nahtloser ins Modell zu integrieren. Zukünftige LLMs könnten von Haus aus "tool-aware" sein, sodass das agentische Reasoning eng mit dem **inneren Prompting** verwoben ist.

- **Reflexion (Self-Reflection):** *Reflexion* (Shinn et al., 2023) ist ein Rahmenwerk, das Agenten befähigt, aus **Fehlern und Feedback** zu lernen, ohne dass dafür Gewichtsupdate oder langwieriges RL-Training nötig ist [57]. Der Kern des Ansatzes: Der Agent verfügt über eine **dynamische Gedächtnisbank**, in die er nach jedem Versuch **sprachliche Reflexionen** ablegt [58]. Wenn eine Aufgabe nicht gelöst wurde oder ein Fehler auftrat, "überlegt" der Agent laut (in Textform), was schiefging und wie er es beim nächsten Anlauf besser machen könnte. Dieses selbstgenerierte Feedback – etwa *„Ich habe den Rechenfehler übersehen, nächstes Mal achte ich darauf"* – wird im **episodischen Gedächtnis** gespeichert [59]. Bei einem neuen Versuch (der Agent startet die Aufgabe erneut) liest er seine früheren Reflexionen mit ins Prompt ein und versucht, diese Ratschläge zu befolgen [59]. Reflexion als Konzept stellt damit eine Art **verbalen**

Reinforcement-Learning-Loop dar [60]: Der Agent erhält keinen numerischen Reward, sondern formuliert sich den Reward als Feedback selbst in Worte. Studien zeigten, dass diese Methode die Erfolgsrate bei kniffligen Aufgaben drastisch steigern kann [61]. Zum Beispiel erreichte ein Reflexion-verbesserter Code-Agent ~91% Erfolg auf einem Coding-Benchmark, gegenüber ~80% ohne Reflexion, sogar besser als GPT-4 ohne diesen Mechanismus [61]. Für das agentische Reasoning ist Reflexion bedeutsam, weil es eine **Selbstkorrektur-Schleife** etabliert: Der Agent evaluiert seine eigenen Aktionen (oder lässt einen separaten *Evaluator*-Agent das tun) [62], generiert Verbesserungsvorschläge und *merkt* sie sich. Entscheidend ist, dass diese Schleife in **Natursprache** stattfindet – was mit LLMs hervorragend funktioniert, da sie Sprache gut verarbeiten. Konzepte wie *Reflection* oder *Hindsight* werden inzwischen von vielen Frameworks unterstützt (z. B. CrewAI oder LangGraph) [63] [59]. So kann man Agenten robuster machen, indem man ihnen erlaubt, bei Misserfolg innezuhalten: *Was habe ich falsch gemacht?*, um dann mit dem neuen Wissen erneut zu starten. Damit verlagert man ein Stück Learning vom Modell-Parameter auf die Laufzeit-Ebene (on-the-fly learning).

- **H-Mem (Hierarchical Memory):** *H-MEM* (Sun & Zeng, 2025) adressiert das Problem der **Langzeitgedächtnis-Verwaltung** in LLM-Agenten [64]. Da LLMs eine begrenzte Kontextlänge haben, muss ein Agent, der z. B. dutzende Dialogrunden oder umfangreiche Erfahrungen ansammelt, entscheiden, was er *im Kopf behält*. Bisherige Ansätze nutzen oft Vektorspeicher (ähnlich einer Datenbank), um vergangene Informationen bei Bedarf ähnlichkeitssuche-basiert abzurufen. H-MEM schlägt nun eine **hierarchische Organisation** der Erinnerungen vor: Informationen werden auf verschiedenen **Abstraktionsebenen** gespeichert [65]. Niedrigste Ebene könnte z. B. konkrete einzelne Dialogturns oder Beobachtungen sein, darüber liegen zusammenfassende Konzepte, und auf höchster Ebene vielleicht thematische Kapitel. Jeder Eintrag erhält **Verweise** (Index-Pointer) auf detailliertere Untereinträge in der nächsttieferen Ebene [65]. Beim **Reasoning** kann der Agent so *top-down* suchen: Erst relevante grobe Kategorie finden, dann zunehmend ins Detail drillen [66]. Dies verhindert, dass bei jeder Frage alle Erinnerungen linear durchsucht werden müssen – was ineffizient wäre. Experimente mit H-MEM zeigten, dass Agents in langen Dialogen konsistenter bleiben und bessere Entscheidungen treffen, wenn sie so ein organisiertes Gedächtnis nutzen [67]. Beispielsweise in einem Rollenspiel-Agenten-Szenario (LoCoMo-Dataset) schlug H-MEM andere Memory-Strategien. Für theoretisches agentisches Reasoning unterstreicht H-MEM die Bedeutung von **Memory Strukturen**: Ein Agent sollte nicht alles flach abspeichern, sondern *vergessen können* bzw. verdichten können, ähnlich wie Menschen Erinnerungen kategorisieren. Das verbessert die **Kontextkohärenz** (der Agent verliert nicht den Faden über viele Turns) und die **Entscheidungsqualität**, weil relevante Fakten schneller auffindbar sind [64]. H-MEM ist somit ein wichtiger Vorstoß, *Langzeit-Reasoning* effizienter zu machen. Konzepte wie **episodisches Gedächtnis**, **semantische Abstraktion** und **Memory Indexing** gewinnen dadurch an Aufmerksamkeit. In Kombination mit Selbstreflexion kann ein Agent so nicht nur denken und handeln, sondern sich auch über lange Zeit *erinnern* und an Erfahrungen anknüpfen.

3. UX-orientierte Agentenarchitekturen und Feedback-gesteuerte Systeme

Neben den rein technischen Aspekten rücken bei Agenten auch **User Experience (UX)** und Interaktionsdesign in den Vordergrund. Ein Agent soll nicht nur *irgendwie* ein Ziel erreichen, sondern dies möglichst im Einklang mit Nutzererwartungen, effizient über mehrere Dialogrunden und unter Wahrung gewisser Verhaltensrichtlinien tun. Hier spielen Konzepte wie **Planning vs. Reactivity**,

Memory Handling, **Value Alignment**, **User Frustration Detection** und **Multi-turn Coherence** eine Rolle.

- **Planning vs. Reactivity:** Ein zentrales Architekturmuster ist die Frage, ob ein Agent vorab einen **Plan entwirft** oder **reaktiv Schritt-für-Schritt** agiert. *Planung* bedeutet, dass der Agent (oder ein Planner-Modul) eine Sequenz von Aktionen vorgedacht hat. Beispielsweise könnte ein Agent, der einen Reiseplan erstellt, zunächst alle erforderlichen Schritte auflisten: *"1. Benutzer nach Reiseziel fragen, 2. Budget erfragen, 3. Flüge suchen, 4. Hotels suchen, ..."*. *Reaktivität* hingegen heißt, der Agent entscheidet **situativ** nach jeder Nutzereingabe, was als Nächstes zu tun ist, ohne einen fixen globalen Plan. Traditionell arbeiten LLM-Agenten reaktiv (siehe ReAct-Muster), was ihnen Flexibilität verleiht – sie können auf unvorhergesehene Nutzerfragen reagieren und den Kurs ändern. Allerdings kann fehlende Planung zu Ineffizienzen führen (z. B. unnötigen Schleifen) oder dazu, dass der Agent sich verzettelt. Neuere **hybride Ansätze** kombinieren beides: So propagieren einige Experten ein *"Plan-and-Execute"*-Paradigma – erst lässt man den Agenten einen groben Plan erstellen, dann werden die einzelnen Schritte reaktiv abgearbeitet. Dadurch bekommt der Nutzer ggf. Einsicht in den Plan und kann korrigieren (bessere UX), und der Agent behält ein *Zielgerüst* im Auge. UX-seitig ist wichtig: **Transparenz und Steuerbarkeit.** Ein planender Agent kann dem User erklären *"Ich werde folgendermaßen vorgehen..."*, was Vertrauen schafft. Ein rein reaktiver Agent wirkt eher wie eine Blackbox, kann aber natürlicher und improvisationsfähiger erscheinen. Viele UX-orientierte Architekturen (z. B. virtuelle Assistenten) bevorzugen eine Mischung: deterministische Unterroutinen für Standardfälle (Planung) und KI-Generativität für Ausnahmen (Reaktivität) [68] [9]. Für Entwickler bedeutet dies, den **Komplexitäts-Trade-off** abzuwägen: Geplante Workflows sind vorhersehbarer (besser testbar, konsistenter Antworten), reaktive Agenten sind anpassungsfähig, aber schwerer zu kontrollieren. Best Practices umfassen, dass Agenten bei komplexen Aufgaben dem Nutzer ihren Plan vorschlagen (*"Soll ich erst X tun und dann Y?"*) – so wird die Interaktion sicherer und der Nutzer fühlt sich einbezogen.

- **Memory Handling (Erinnerungsverwaltung):** UX-technisch enorm wichtig ist, wie der Agent mit **Konversationsverlauf und Wissen** umgeht. Ein häufiges Problem ist, dass Agenten in längeren Dialogen **Kontext vergessen** oder widersprüchlich werden, weil die Vergessensgrenze (Kontextfenster) erreicht ist. Gute Agentenarchitekturen implementieren daher **Memory-Mechanismen**: Von **Kurzzeitgedächtnis** (z. B. Speicherung der letzten N Dialogturns) bis **Langzeitgedächtnis** (Persistieren von Fakten über Sessions hinweg) [69]. Für die UX bedeutet dies: Der Agent kann sich an frühere Aussagen des Nutzers erinnern (*"Wie schon erwähnt, ist Ihr Budget 1000€"*) und vermeidet redundante Fragen. Technisch kommen oft Vektor-Datenbanken zum Einsatz, um vergangene Äußerungen semantisch auffindbar zu machen. Neuere Ansätze wie das bereits erwähnte H-MEM strukturieren Erinnerungen mehrstufig, um **gezielt relevante Details** abrufen zu können [65]. **Summarization** ist ein weiteres Tool: Der Agent fasst alte Gespräche zusammen, um Platz im Kontext zu sparen, behält aber Kerninfos. Ein UX-Aspekt ist auch **Memory Editability**: Manche Systeme erlauben dem Nutzer, Teile des Agenten-Gedächtnisses zu löschen oder zu korrigieren (z. B. "Vergiss das vorige Kommando"). Dies kann wichtig sein, falls der Agent auf einer falschen Annahme beharrt. Des Weiteren müssen Architekten beachten, *welche* Infos wie lange gespeichert werden – aus Datenschutz und Relevanzgründen. Im Kontext von **Multi-turn Coherence** (siehe unten) ist Memory Handling zentral: Ein Agent mit gutem Memory wird über viele Turns hinweg konsistent antworten können, weil er die Gesprächshistorie und seine eigenen früheren Aussagen berücksichtigt [64]. In der Interaktion wirkt sich das aus als **kohärente Persönlichkeit** und **Themenstetigkeit**.

- **Value Alignment (Wertausrichtung & Verhalten):** Ein UX-orientierter Agent muss auch auf die **Bedürfnisse, Erwartungen und Werte** des Nutzers (und der Entwicklerorganisation)

ausgerichtet sein. Darunter fallen ethische und stilistische Aspekte: z. B. **Höflichkeit**, **Bias-Vermeidung**, **keine toxischen Äußerungen**, aber auch **Markenkonformität** (Tonfall passend zur Marke). Technisch wird Value Alignment häufig durch *System Prompts* und **Richtlinien** erreicht (z. B. die OpenAI- oder Anthropic-Modelle haben in ihren System-Einstellungen moralische Leitplanken). Ein Beispiel ist Anthropics *Constitutional AI*, wo dem Modell bestimmte Prinzipien (wie Nicht-Diskriminierung) eingepflanzt wurden. Auf Architekturebene kann man Value Alignment durch einen separaten **"Guardrail"-Agenten** umsetzen – so hat AutoGen z. B. den *Safeguard*-Agent, der Outputs prüft [27]. Auch **Tool-Zugriffsrestriktionen** sind Teil des Alignment: Der Agent sollte nur das tun, was erlaubt ist (z. B. keine internen Daten leaken). Für die UX wichtig ist auch **Erklärbarkeit**: Ein alignierter Agent sollte dem User gegenüber transparent sein, warum er eine Frage nicht beantworten kann (statt einfach "Nein" zu sagen, lieber *"Ich darf hierzu keine Auskunft geben, weil…"*). **Frustrationsvermeidung** hängt eng damit zusammen – ein Agent, der die *Intention* des Nutzers wertschätzt, wird versuchen, notfalls alternative Lösungen anzubieten, statt nur Richtlinien durchzusetzen. Value Alignment umfasst ferner **Persönlichkeitsabstimmung**: Ein Finanz-Chatbot z.B. sollte sachlich und vertrauenswürdig klingen, ein Kinder-Lernassistent freundlich und einfach. Dies erreicht man oft durch sorgfältiges Prompt Design (Stimmvorgaben) oder durch Feintuning auf passenden Daten. Zusammengefasst sorgt Value Alignment dafür, dass der Agent **verlässlich und vertrauenswürdig** rüberkommt, was die Nutzerakzeptanz steigert. Da LLM-Agenten dazu neigen können, manchmal "abzudriften" (halluzinieren, unangebrachte Witze etc.), ist dies ein aktives Forschungsfeld. Technische Lösungen wie **Nachbearbeitung** von Model Outputs (through moderation APIs) werden oft kombiniert mit präventiven Anweisungen im Prompt.

- **User Frustration Detection:** In längeren Interaktionen kann es passieren, dass der Nutzer unzufrieden oder genervt reagiert – sei es, weil der Agent die Frage nicht versteht, falsche Antworten gibt oder zu langsam ist. Moderne Systeme versuchen daher, **Nutzerfrustration automatisch zu erkennen**, um gegenzusteuern. Dies geschieht meist durch **Sentimentanalyse** oder Mustererkennung im User-Input [70]. Beispiele: CAPS LOCK, viele "!" oder Sätze wie *"Du verstehst mich einfach nicht"* deuten auf Frust hin. Ein Task-Oriented Dialogue System könnte die letzten Turns analysieren und einen *Frustration Score* berechnen [70]. Es gibt auch Ansätze, in denen ein zweites Modell (oder ein bestimmter Prompt mit dem gleichen Modell) bewertet: *"Hat der Nutzer am Ende Frustration gezeigt – ja oder nein?"* [71] [72]. Wenn Frustration erkannt wird, kann die **Agentenarchitektur adaptiv reagieren**: z. B. Tonfall ändern, proaktiver um Klarstellung bitten oder höflich vorschlagen, einen menschlichen Mitarbeiter hinzuzuziehen. Ein praktisches Beispiel: Ein Support-Chatbot registriert, dass der Kunde Sätze schreibt wie *"Das hilft mir alles nicht weiter"* in gereiztem Ton. Daraufhin könnte der Bot in den *Deeskalationsmodus* wechseln – etwa: *"Entschuldigung, dass Sie noch keine Lösung haben. Lassen Sie es uns anders versuchen: …"* oder *"Ich verstehe Ihre Frustration. Soll ich Ihren Fall an einen Menschen übergeben?"*. Solche Mechanismen erhöhen die **Nutzungszufriedenheit**, da sie zeigen, dass der Agent die Emotionen des Users *merkt* und empathisch reagiert. Technisch ist das keine einfache Herausforderung: Ironie oder verdeckte Frustration zu erkennen, erfordert fortgeschrittene NLP. Allerdings sind LLMs an sich schon recht gut im Ton-Verständnis, sodass ein Zero-shot-Prompt ("Ist der User frustriert?") oft brauchbare Ergebnisse liefert [73] [72]. Wichtig in der Architektur ist, dass diese Bewertungsschritte schnell geschehen (Latenz!) und robust sind. Einige Frameworks wie Arize AX haben vordefinierte Evaluatoren für "User Frustration" entwickelt [71] [72]. UX-seitig sollte der Agent niemals schnippisch oder genervt zurück reagieren – selbst wenn der Nutzer unfreundlich ist. Das fällt ebenfalls unter diese Designkomponente.

- **Multi-turn Coherence:** Ein großer Test für Agenten ist die **Kohärenz über lange Dialoge** hinweg. Das heißt: Bleibt der Agent beim Thema? Widerspricht er sich nicht selbst? Erinnert er sich an Details, die der Nutzer erwähnt hat, auch nach 10 Zwischenfragen? Multi-turn Coherence

wird durch Kombination mehrerer der oben genannten Faktoren erreicht. **Memory** spielt die Hauptrolle – ohne Langzeitgedächtnis keine Kohärenz. Aber es geht auch um **internes konsistentes Verhalten**: Ein Agent sollte idealerweise eine konstante Persönlichkeit haben und nicht bei jeder Antwort den Stil wechseln (außer der Nutzer wünscht das explizit). UX-orientierte Architekturen geben oft einen *Persona-Prompt* vor (z. B. System: "Du bist ein höflicher Reiseassistent…"), der in jeder Antwort mitschwingt. Ebenso gehört dazu, **Benutzereingaben richtig zu interpretieren**, auch wenn sie Bezug auf Vergangenes nehmen (*"sowas wie vorhin"* sollte der Agent verstehen können). Technisch kann Multi-turn Coherence durch **Zwischenspeicherung von Zwischenerkenntnissen** verbessert werden. Einige Agenten führen eine *Knowledge Base* mit Fakten, die im Gespräch geklärt wurden (z. B. Nutzerpräferenzen). Diese kann in späteren Antworten referenziert werden, ohne jedes Mal neu gefragt zu werden. Ein wichtiger Spezialfall ist **Co-reference Resolution**: Wenn der Nutzer sagt *"Und was ist mit dem dritten Angebot?"*, muss der Agent aus dem Verlauf wissen, was "das dritte Angebot" war. Solche Fähigkeiten kann man entweder direkt vom LLM (das oft erstaunlich gut Coreference auflösen kann) erwarten, oder durch explizite Programm-Logik (Kontextfenster mit Annotationsspeicher). **Kohärenz** bezieht sich auch auf das Vermeiden von Widersprüchen: Es ist verwirrend, wenn der Agent in Turn 5 etwas behauptet, was Turn 2 bereits negiert wurde. Hier können **Consistency-Checks** helfen – z. B. ein paralleler Prozess, der Antworten gegen vorherige Fakten prüft. Allerdings ist das schwierig vollautomatisch umzusetzen. Häufig genügt es, wenn der Agent dank ausreichendem Memory und gutem Prompting *selbst* konsistent bleibt. Experimente mit Hierarchical Memory (siehe H-MEM) zeigen, dass strukturierte Speicher die Dialogkohärenz signifikant verbessern, weil relevante Kontextinfo besser gefunden wird [74] [67]. Aus Nutzersicht ist ein kohärenter Agent Gold wert: Er vermeidet wiederholtes Nachfragen, liefert logische, aufeinander aufbauende Antworten und fühlt sich "aus einem Guss" an. Das steigert Vertrauen und das Gefühl, mit einem kompetenten Gegenüber zu sprechen, statt mit einem zufälligen Antwortgenerator. Zusammenfassend erreichen UX-orientierte Architekturen Multi-turn Coherence durch a) **effizientes Kontext- und Wissensmanagement**, b) **einheitliche Personas/ Styles**, und c) optional **Post-Processing** (z. B. consistency filters).

All diese UX-Aspekte – Planung vs. Reaktion, Memory, Alignment, Frustrationserkennung, Mehrturn-Kohärenz – greifen ineinander. Ein konkretes Beispiel zum Abschluss dieses Abschnitts: Unser Auto-Berater-Chatbot aus der Einleitung. In einem im *Optimierungsbedarf_AutoBot.txt* gezeigten Chat sieht man, wie der Nutzer erst vage einen "robusten Dauerläufer" sucht, der Bot ein paar Autos vorschlägt (u.a. einen Porsche), der Nutzer irritiert reagiert, warum ein teurer Porsche genannt wurde, und Frustration zeigt *"Ich glaube, du kannst mir nicht wirklich weiterhelfen"*. Hier müsste die Agenten-UX ansetzen: Der Bot sollte a) erkennen, dass der Nutzer unzufrieden ist (negativer Satz), b) sich entschuldigen und erklären, warum der Porsche auftauchte (was er tat: *"Verstehe Verwirrung… nicht passend zu robustem Dauerläufer…"*), c) sein Vorgehen anpassen – ggf. den Plan ändern und neue Modelle vorschlagen, die wirklich zum "Dauerläufer"-Kriterium passen, d) idealerweise gelernt haben, dass die Gewichtung der Kriterien falsch war (der Bot hat "Fahrspaß" überbewertet). Dieses Beispiel zeigt, wie eng Planung (die Gewichtung, welches Auto vorschlagen), Memory (merken was *"robust"* bedeutet für den User), Alignment (Eingehen auf Budget, kein Luxussportwagen anbieten, wenn Nutzer das implizit ausschließt) und Frustrationserkennung verzahnt sind. Eine **UX-orientierte Architektur** würde hier evtl. einen *Reflexionsschritt* einbauen: Der Bot reflektiert intern *"Oops, Porsche war ein Fehlgriff, da Nutzer Robustheit meint und Budgetbedenken hat"* und justiert seine Empfehlungslogik. Dies könnte etwa durch *wertbasierte Reranking* der Angebotsempfehlungen oder Einbeziehung eines **User Profiling**-Moduls (kennt Budgetpräferenzen) erreicht werden [75] [76]. Insgesamt verdeutlicht das: Die Technik allein reicht nicht – die Agenten müssen auch *menschlich* und adaptiv agieren.

4. Prompting-Techniken für "Vibe-Coding"

Unter *Vibe-Coding* versteht man das **Steuern von LLM-Ausgaben durch die implizite Atmosphäre des Prompts**, statt durch wörtliche Anweisungen zur Ausgabeform. Der Begriff wurde Anfang 2025 von Andrej Karpathy geprägt und beschreibt, wie man gewünschte **Stile, Tonalitäten und Intentionen** in die KI-Antwort *hineincodiert*, indem man dem Modell gewisse *Vibes* (Stimmungen) vorgibt [77] [78]. Anders gesagt: Statt dem Modell explizit zu sagen *"Schreibe in formellem Ton"* oder *"Sei lustig"*, gestaltet man den Prompt (System- oder Usernachricht) so, dass solche Eigenschaften **zwischen den Zeilen** mitschwingen [79].

Was ist Vibe-Coding genau? Laut einer Definition *"ist Vibe-Coding die Praxis, stilistische Signale in Prompts einzubetten, um zu beeinflussen, wie ein KI-Modell antwortet"* [79]. Dabei werden **Wortwahl**, **Satzbau**, **Punktuation** und sogar **kulturelle Anspielungen** genutzt, um dem Modell einen gewissen *Gefühlston* zu vermitteln [79]. Das Modell erkennt diese subtilen Muster und stimmt seine Antwort darauf ein, da es während des Trainings ähnliche Stilzusammenhänge gelernt hat [80]. Ein Beispiel: Soll ChatGPT einen Code erklären, kann ich entweder trocken fragen *"Erkläre den folgenden Code."* – oder mit Vibe-Coding: *"Stell dir vor, du sitzt mit einem Kollegen beim Kaffee und erklärst ihm locker den Code…"*. Letzteres Prompt erzeugt meist eine lockerere, zugänglichere Erklärung [81]. Der *Inhalt* (Code erklären) bleibt gleich, aber der *Vibe* (Kaffeegespräch unter Kollegen) differiert deutlich von einer formalen Anweisung. Das Modell antizipiert dann einen informellen Ton, vielleicht mit etwas Humor, aber dennoch technisch korrekt – genau was man wollte.

Techniken und Best Practices: Über die letzten Monate hat sich ein Set an Techniken herauskristallisiert, um Vibe-Coding effektiv einzusetzen [82]:

- *Style Priming:* Wörter transportieren Ton. Durch gezielte Adjektive und Formulierungen kann man das Modell auf einen Stil eichen [83]. Z.B. *"Bitte antworten Sie prägnant und scharf."* vs. *"Bitte antworten Sie fließend und umgangssprachlich."* – ersteres primt das Modell auf eine kurze, direkte Sprache, letzteres auf längere, lockere Sätze [83]. Auch Domain-Jargon gehört hier dazu: Fachbegriffe signalisieren Fachstil, Jugendsprache signalisiert lockeren Ton.

- *Context Seeding:* Die Einbettung der Anfrage in einen bestimmten **Kontext oder ein Szenario** verändert den Vibe [84]. Wenn ich z. B. schreibe *"Du bist ein strenger Professor, der… erläutert"*, erwarte ich einen belehrenden, formalen Ton. Wenn ich hingegen schreibe *"Stell dir vor, wir brainstormen in einer Bar über dieses Problem"*, wird die Antwort vermutlich kreativer und ungezwungener. Das **Framing** (Rahmensetzung) ist ein mächtiges Mittel: *"Schreibe in Form eines Märchens…"* oder *"Diskutiere, als ob es eine hitzige Debatte im Parlament ist"* – solche Kontexte beeinflussen Format, Wortwahl und Dynamik der Antwort massiv [84].

- *Persona Signaling:* Man gibt dem Modell eine **Rolle oder Identität**, die es bei der Antwort einnehmen soll [85]. Beispiele: *"Antworte als wärest du ein erfahrener Arzt"*, *"als leidenschaftlicher Umweltaktivist"*, *"als knallharter CEO"*. Diese Personas bringen automatisch bestimmte Tonalitäten mit sich – der Arzt wird sachlich und beruhigend erklären, der Aktivist emotional appellieren, der CEO kurz angebunden und entscheidungsfreudig argumentieren. Wichtig ist dabei, dass man keine expliziten verbalen Ticks vorgibt, sondern das Modell selbst die Register findet. Persona-Priming hilft auch, **Zielgruppen-gerecht** zu formulieren (z. B. *"Erkläre es wie einem fünfjährigen Kind"* lässt einfache Sprache folgen). Eine Variation davon ist *"Imitiere den Stil von [berühmte Person/Autor]"* – hier ist Vorsicht geboten (rechtlich/ethisch je nach Person), aber technisch kann das Modell dann gelernte Stilmerkmale jener Persönlichkeit adaptieren.

- *Emotionale Signale:* Sogar **Emojis, Ausrufezeichen, Ellipsen** etc. können den Vibe beeinflussen. Ein *"Wow, das ist ja großartig..."* zu Beginn eines Prompts stimmt das Modell euphorisch. Ein *"(seufzt)"* kann Resignation signalisieren. Solche indirekten Hinweise nutzt man gezielt, wenn man z. B. eine begeisterte Antwort will oder eine sarkastische. Natürlich ist hier Feingefühl gefragt – zu viele Emoji oder übertriebene Interjektionen könnten das Modell ablenken oder unerwünschte Stile erzeugen.

- *Shadow Prompting:* Dieser Begriff wird manchmal verwendet, um **verdecktes Prompting** zu beschreiben – d. h. Anweisungen, die das System erhält, ohne dass der Endnutzer sie sieht [86]. Im Kontext Vibe-Coding kann das bedeuten, dass Entwickler im System-Prompt einen bestimmten Stil verankern (z. B. "Immer freundlich und hilfsbereit bleiben, egal wie der Nutzer schreibt"). Dieser *Schatten-Prompt* beeinflusst dann jede Antwort, auch wenn der Nutzer es nicht merkt. Aus UX-Sicht ist das zweischneidig: Es sorgt für Konsistenz im Ton, aber der Nutzer kann die *Biases* und Einschränkungen nicht sehen [86]. Dennoch ist Shadow Prompting verbreitet, etwa um **Markenstimmen** sicherzustellen. Best Practice ist, die Shadow Prompts schlank zu halten (sie gehen ja vom Kontextfenster ab) und klar zu definieren, welche stilistischen Leitplanken gelten. Beispiel: Ein Unternehmens-Chatbot hat immer unsichtbar eingebettet: *"Du bist der Assistent von Firma X, sprich Kunden mit Sie an, bleibe positiv und lösungsorientiert."* – So wird ein einheitlicher *Corporate Vibe* garantiert.

- *Style Examples:* Manchmal hilft ein **Beispielabsatz** im gewünschten Stil. Etwa: *"Antwort-Beispiel (freundlich, locker): 'Hey du! Also, ich hab mir mal Gedanken gemacht..."*. Danach die Aufforderung: *"Jetzt du:"*. Das Modell imitiert dann oft den Stil des Beispiels. Dies ist analog zum Few-Shot-Learning, nur dass der Schwerpunkt auf Stil statt Inhalt liegt. Dieses *Style Priming via Example* kann Teil des System-Prompts sein (versteckt) oder explizit dem Nutzer prompt folgen.

Vibe-Coding vs. klassische Prompting-Techniken: Wichtig zu betonen ist, dass Vibe-Coding **keine Ablösung**, sondern **Ergänzung** zum herkömmlichen Prompt Engineering darstellt [87]. Ein guter Prompt hat oft beides: *Explizite Struktur* und *impliziten Vibe*. Beispiel: *"Erstelle eine Liste von 3 Optionen in einem freundlichen Ton, als würdest du einem Kollegen raten."* – Hier ist "Liste von 3 Optionen" eine klare Vorgabe (strukturierter Output), während "freundlicher Ton, wie einem Kollegen" den Vibe setzt. Die Mischung macht's: Explizite Formatwünsche sorgen für kontrollierbare Ausgaben (z. B. JSON, Listen, Titel etc.), während implizite Vibes die Antwort natürlicher und überzeugender wirken lassen [88]. Vibe-Coding erhöht oft die **Robustheit** der Antworten [89]: Wenn mal eine strikte Anweisung nicht 100% befolgt wird, fängt der Vibe trotzdem viel ab. Beispiel: Statt "Mache Aufzählung" kann man vibe-basiert sagen "Lass uns das Schritt für Schritt anschauen...". Selbst wenn das Modell keine Zahlen voranstellt, kommt meist ein schön gegliederter Text dabei heraus, da "Schritt für Schritt" diesen Eindruck imitiert.

Beispiele: Im Bereich *Coding mit LLM* – worauf der Begriff Vibe-Coding ursprünglich anspielte – kann man mit dem Prompt-Ton enorm viel bewirken. Ein interessanter Vergleich aus einem Artikel [90] [91]:

- **Formal-akademischer Vibe (ChatGPT):** *"I require a function that computes the Fibonacci sequence utilizing recursion with memoization for optimization. Please implement this in Python with appropriate documentation."* – Das ergibt als Antwort sehr sauberen, ordentlich dokumentierten Code mit Docstrings, Type Hints und klarer Struktur [92] [93].

- **Verspielter, informeller Vibe (ChatGPT):** *"Hey, kannst du mir flott 'ne Python-Funktion basteln, die Fibonacci macht? Vielleicht rekursiv, aber nicht super langsam? Danke!!"* – Hierauf liefert das Modell denselben Codezweck, aber mit lässigen Kommentaren ("// The magic happens here") und einem lockeren Ton im Code (keine Docstring-Orgie, dafür usage-Beispiel mit umgangssprachlichem

Kommentar) [91] [94]. Die Funktion heißt dann z.B. `fib` statt `fibonacci_memoized`, und Kommentare sind eher humorvoll.

Dieses Beispiel zeigt: Beide Male wurde *nicht explizit* gesagt "schreibe es formell/locker", aber die Prompt-Formulierung hat implizit den Stil vorgegeben – *das ist Vibe-Coding in Aktion*.

Weitere Anwendungen: Vibe-Coding ist in vielen Bereichen nützlich [95] [96]. In der **Bildung** kann ein Lehrer-Avatar durch Vibe-Coding angepasst werden: *"Erkläre es, als würdest du einem neugierigen Teenager etwas Spannendes erzählen"* – das verhindert trockene Wikipedia-Antworten und fördert Engagement. In der **Kreativarbeit** (Story schreiben, Marketing) kann man das Modell auf Stilreisen schicken, z. B. *"Beschreibe den Sonnenuntergang wie ein 19. Jahrhundert Poet"* für blumige Sprache [97]. In **Business-Kommunikation** hilft Vibe-Coding, den richtigen Ton zu treffen: *"Formuliere die E-Mail in einem beruhigenden, professionellen Ton, der die Bedenken des Kunden ernstnimmt."* – So erhält man eher empathische, aber sachliche Texte [98]. Wichtig ist immer, dass der *implizite Tonwunsch* konsistent ist und zum Ziel passt.

Stolperfallen: Auch beim Vibe-Coding gibt es Herausforderungen [99]. Ein häufiger Fehler ist, zu viele widersprüchliche Vibes zu mischen (*"Sei witzig, aber ernst, und sprich jugendlich, aber professionell"* – das verwirrt das Modell und resultiert in unklarem Stil). Besser ist, einen klaren primären Vibe zu setzen. Ein weiterer Punkt: Manche Modelle können Vibes falsch interpretieren – insbesondere kulturelle Referenzen (*"Kling wie Yoda"* führt z.B. bei kleineren Modellen evtl. nur zu Kauderwelsch). Hier hilft es, in kleinen Schritten auszuprobieren. Zudem sollte man bedenken, dass Vibe-Coding indirekt ist – es garantiert nicht 100% Kontrolle. Wenn exakte Formatierung nötig ist, nicht nur auf Vibe vertrauen. **Überladung des Prompts** mit stilistischen Floskeln kann auch Kontraproduktiv sein: Das Modell konzentriert sich dann mehr auf den Ton als auf den Inhalt. Balance finden ist hier die Kunst.

Abschließend: Vibe-Coding hat sich als **praktische Fertigkeit** für Prompt-Ingenieure etabliert, um die *"Hidden Layer"* der Modellgenerierung – nämlich Stimmung und Kontext – zu beeinflussen [100] [101]. Gerade im Zusammenspiel mit agentischen Workflows ist dies relevant: Man kann System-Prompts nutzen, um z.B. dem Planer-Agenten einen analytischen, straffen Ton zu geben, während der Endnutzer-facing Agent einen empathischen Ton erhält. So wirken Multi-Agent-Systeme nach außen konsistent und aufgabenangemessen. Vibe-Coding macht KI-Ausgaben oft **menschennäher, abwechslungsreicher und zielgruppenpassender**, was in vielen Anwendungen entscheidend für die Akzeptanz ist.

5. Literaturübersicht und Ressourcenempfehlungen

Zum Vertiefen der oben behandelten Themen folgt eine kommentierte Auswahl aktueller Papers, Artikel und Framework-Dokumentationen (Stand 2023–2025). Jede Quelle ist mit Titel, Autoren/Team, Jahr sowie einer Kurzbeschreibung und dem Bezug zu den obigen Themen aufgeführt:

1. **ReAct: Synergizing Reasoning and Acting in Language Models** – *Shunyu Yao et al., 2022 (ArXiv preprint)* [52] [53].
   Beschreibung: Introduziert das ReAct-Paradigma, bei dem LLMs durch abwechselndes **Denken (Reasoning)** und **Handeln (Tool-Aufrufe)** komplexe Aufgaben lösen. Zeigt empirisch, dass ReAct sowohl auf Wissensfragen als auch Entscheidungsproblemen reine "Act-only" oder "Thought-only" Ansätze übertrifft.
   Relevanz: Grundlegendes Konzept für viele Agenten-Frameworks (z. B. LangChain Agents) [52]. Legt die theoretische Basis für agentisches Reasoning im Schritt-für-Schritt-Stil und inspiriert Kombinationen von Kettenlogik mit Tool-Integration.

2. **Toolformer: Language Models Can Teach Themselves to Use Tools** – *Timo Schick et al., 2023 (Meta AI, ArXiv)* [56].
   Beschreibung: Präsentiert ein Verfahren, um ein Sprachmodell so zu trainieren, dass es **selbstständig API-Calls einbettet**. Das Modell entscheidet, *wann* und *welche* externen Tools (Rechner, Suchmaschine, Übersetzer, Kalender etc.) aufzurufen sind und wie die Ergebnisse in den Text eingebaut werden [56]. Erreicht bessere Zero-Shot-Ergebnisse auf Aufgaben wie Rechnen und Wissensfragen, ohne die generelle Sprachfähigkeit zu verschlechtern.
   Relevanz: Zeigt einen Weg zur **nativen Tool-Nutzung** durch LLMs, was für agentische Workflows entscheidend ist. Theoretische Untermauerung für Ansätze, bei denen das Modell den Gebrauch von Tools als Teil seines Outputs plant (Vorbild für Integration von API-Aufrufen in Ketten).

3. **CAMEL: Communicative Agents for "Mind" Exploration of LLM Society** – *Guohao Li et al., 2023 (NeurIPS 2023)* [54] [55].
   Beschreibung: Untersucht **autonome Kooperation** mehrerer KI-Agenten via *role-playing*. Stellt ein Framework vor, in dem zwei ChatGPT-ähnliche Agenten (z.B. *AI User* und *AI Assistant*) mit vordefinierten Rollen und Zielen miteinander chatten, um Aufgaben zu lösen. Nutzt *Inception Prompting* (Systemprompts mit Rollenbeschreibung), um die Interaktion auf Kurs zu halten [55]. Bietet Einblicke in die "kognitiven" Prozesse einer Multi-Agent-"Gesellschaft" und generiert Daten, um deren Verhalten zu analysieren.
   Relevanz: Wegweisend für **Multi-Agent-Systeme**. Zeigt, wie man durch clevere Prompt-Setups Agenten kollaborieren lassen kann, was z.B. CrewAI und AutoGen (mit Commander/Writer) praktisch einsetzen. Hilft zu verstehen, wie Rollenverteilung und Kommunikation die Problemlösung beeinflussen.

4. **Reflexion: Language Agents with Verbal Reinforcement Learning** – *Noah Shinn et al., 2023 (ArXiv preprint, NeurIPS 2023 Workshop)* [57] [58].
   Beschreibung: Führt das *Reflexion*-Framework ein, bei dem Agenten mittels **sprachlichem Feedback** aus vorherigen Versuchen lernen, statt via Gradientenabstieg. Der Agent reflektiert in eigenen Worten über Fehler oder Misserfolge und speichert diese Reflexion in einer Memory, um beim nächsten Anlauf bessere Entscheidungen zu treffen [57] [59]. Demonstriert signifikante Leistungsverbesserungen auf diversen Aufgaben (u.a. Coding/HumanEval: 91% zu 80% gegenüber GPT-4 ohne Reflexion) [61].
   Relevanz: Schlüsselwerk für **Selbstkorrektur** in Agenten. Das Konzept dynamischer *Episodic Memory* und Self-Feedback wurde in vielen Systemen aufgegriffen (z.B. LangChain, Generic "Reflection" Pattern [63]). Erhöht Zuverlässigkeit agentischer Workflows durch iterative Verbesserung und verbindet Reinforcement Learning mit In-Context-Lernen auf neuartige Weise.

5. **Hierarchical Memory for High-Efficiency Long-Term Reasoning in LLM Agents** – *Haoran Sun, Shaoning Zeng, 2025 (ArXiv)* [65] [66].
   Beschreibung: Forschungsarbeit, die eine **hierarchische Gedächtnisarchitektur (H-MEM)** vorschlägt. Speichert Agenten-Erinnerungen auf mehreren Abstraktionsebenen und versieht jede Memory-Einheit mit einem Index zu detaillierteren Untereinheiten [65]. Durch einen indexbasierten Abruf kann der Agent bei Bedarf effizient von allgemeinen Zusammenfassungen zu spezifischen Details "navigieren", statt alle Erinnerungen durchsuchen zu müssen [66]. Übertrifft in Tests (LoCoMo-Dialogaufgaben) klassische Vektorstore-Methoden in Bezug auf Dialogkonsistenz und Erfolg [67].
   Relevanz: Adressiert das **Langzeit-Kontext**-Problem für Agenten. Liefert Ideen für Memory-Verwaltung in UX-orientierten Architekturen, insbesondere für **Multi-turn Coherence** über Dutzende von Turns. Kann in praktisch allen Agentenframeworks ergänzend eingesetzt werden, um Wissens- und Kontextverwaltung zu optimieren.

6. **AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation** – *Qingyun Wu et al., 2023 (Microsoft Research, ArXiv)* [26].
   Beschreibung: Stellt das Microsoft *AutoGen*-Framework vor. Beschreibt die Fähigkeit, mittels **mehrerer conversable Agents** komplexe Anwendungen zu bauen, wobei Agents unterschiedliche Rollen einnehmen können (z.B. Nutzer, Assistent, Tool-Executor) [26]. Hebt die **Flexibilität** hervor: AutoGen-Agents können mit anderen LLMs, menschlichen Inputs und Tools interagieren, und Entwickler können die Interaktionsmuster in natürlicher Sprache oder Code definieren [26]. Empirische Beispiele aus Mathematik, Coding, Entscheidungsfindung etc. illustrieren die breite Anwendbarkeit.
   Relevanz: Praktisches Framework für viele Ideen aus Sektion 1 & 2. Insbesondere die **Agentenkommunikation** und die Mischung aus deterministischen und dynamischen Abläufen werden hier realisiert. Die Publication dient als Brücke zwischen theoretischen Ansätzen (z.B. CAMEL, Reflexion) und einer nutzbaren Bibliothek. Hilfreich, um zu sehen, wie Microsoft komplexe Agentensysteme (mit Asynchronität, verteilten Agents) umsetzt.

7. **TaskWeaver: A Code-First Agent Framework** – *Bo Qiao et al., 2024 (Microsoft, ArXiv)* [44].
   Beschreibung: Beschreibt *TaskWeaver*, ein Framework, das LLM-Agenten einbettet, indem es Nutzeranfragen in **ausführbaren Code** übersetzt [45]. Stellt heraus, dass TaskWeaver reichhaltige Datenstrukturen und dynamische Plugin-Nutzung unterstützt, wodurch LLMs komplexe Logik durch Code generieren und ausführen können [44]. Außerdem betont es sichere Ausführung (z.B. durch Sandboxing) und Einbringung von domain-spezifischem Wissen via Codebeispiele.
   Relevanz: Speziell relevant für **agentic reasoning in technischen Domains** (Analytics, Scripting). Es untermauert, dass LLMs als *Coder* agieren können – was modulare und überprüfbare Workflows erlaubt (Code kann getestet werden). Passt in den Trend, LLMs mit traditionellem Programmieren zu verbinden, um Zuverlässigkeit zu erhöhen. Für Entwickler, die mit Tools wie LangChain arbeiten, bietet TaskWeaver einen alternativen Denkansatz, wo der Agent quasi zum "Programmierer seiner eigenen Lösungen" wird.

8. **Generative Agents: Interactive Simulacra of Human Behavior** – *J. S. Park et al., 2023 (Stanford, Proc. ACM UIST)* [102].
   Beschreibung: Präsentiert *Generative Agents*, KI-basierte Simulations-Agenten, die in einer virtuellen Welt agieren (bekannt geworden durch die "Stanford Sims"-Demo einer Kleinstadt voller KI-Charaktere). Jeder Agent hat Gedächtnis, tägliche Pläne und kann mit anderen interagieren, wodurch emergentes sozial-koherentes Verhalten entsteht. Die Arbeit zeigt, wie durch **Memory (Kurz- und Langfrist)**, **Planning (Tagesablauf generieren)** und **Reflection (Selbstgespräche, um Wichtiges zu extrahieren)** äußerst realistische, konsistente Charaktere modelliert werden können.
   Relevanz: Dieses Paper verbindet **agentisches Gedächtnis, Planen und Interaktion** beispielhaft. Obwohl die Domäne "Simulierte Menschen in Sandbox" ist, gelten die Prinzipien allgemein: für Multi-turn Coherence und Langzeitverhalten sind Gedächtnis und regelmäßige Reflexion essentiell. Entwickler von Chatbots können daraus lernen, wie man Personas persistent hält und wie Agenten eigenständig **hochkoherente Multi-Turn-Dialoge** führen können. Es inspiriert zudem UX-Aspekte (z.B. wie Agenten auf externe Ereignisse reagieren und eigene Ziele verfolgen können).

9. **A Review on Vibe Coding: Fundamentals, State-of-the-art, Challenges and Future Directions** – *Partha Pratim Ray, 2025 (TechRxiv Preprint)* [103].
   Beschreibung: Umfangreiches Übersichts- und Positionspapier, das den Begriff *Vibe-Coding* systematisch aufarbeitet. Definiert die Grundlagen (emotionale und tonale Intention in Prompts), fasst aktuelle Praktiken und Fallstudien zusammen und diskutiert Herausforderungen

(z. B. Missverständnisse, Balance zwischen Instruktion und Kreativität). Auch Sicherheit und ethische Aspekte von Vibe-Coding (z.B. versteckte Manipulation durch Shadow Prompts) werden beleuchtet.
Relevanz: Ideal, um das Thema *Vibe-Coding* im Gesamtzusammenhang zu verstehen. Deckt vieles ab, was in Sektion 4 angerissen wurde, mit zusätzlichem akademischen Auge. Gerade wer im Bereich **Prompt Engineering** tiefer einsteigen will (auch über Coding-Anwendungen hinaus), findet hier einen aktuellen Forschungsüberblick. Für die Praxis liefert es zudem eine Einordnung, welche **Best Practices** sich etabliert haben und wo Grenzen liegen.

10. **LangChain + LangGraph Documentation & Blog** – *Harrison Chase und LangChain Team, 2023–2024* [2] [10].
Beschreibung: Die offizielle Dokumentation von LangChain (insb. zum Modul LangGraph) und Blogartikel wie "How to think about agent frameworks" bieten praxisnahe Einblicke in den Aufbau von agentischen Anwendungen. Es werden grundlegende Patterns diskutiert – z.B. Workflows vs. Agents, Reliability-Herausforderungen, Kontext-Engineering – und erläutert, wie LangGraph diese adressiert. Enthält auch Tutorials und Beispiele zum Erstellen von Graph-Workflows, Persistenz, Memory etc.
Relevanz: **Hands-On-Quelle** für Entwickler. Verknüpft die theoretischen Konzepte (Agents, Tools, Memory, Reflexion) mit konkreter Implementierung in einem populären Framework. Insbesondere die Persistenz- und Debugging-Features von LangGraph [5] [10] zeigen, wie man Zuverlässigkeit in agentischen Systemen erhöht – ein zentrales Anliegen bei Produktionseinsatz. Auch für Prompting liefert LangChain Doku wertvolle Tipps (z.B. *Context engineering* in Agents [104]). Insgesamt ein Muss, um von Theorie zur Praxis zu gelangen und typische Stolpersteine beim Entwickeln *real-world* Agenten zu verstehen.

Diese Auswahl soll sowohl theoretische Grundlagen als auch praktische Frameworks abdecken, die im Bereich agentische Workflows und Vibe-Coding den aktuellen Stand der Technik repräsentieren. Weiterführend lohnt sich ein Blick in angrenzende Surveys (z.B. *"Generative to Agentic AI: Survey..."*) und die lebhafte Community-Diskussion (Blogs, Foren wie reddit r/LLM, r/AgentGPT), um stets auf dem Laufenden zu bleiben, da sich dieses Feld in rasantem Tempo weiterentwickelt.

---

[1] [2] [3] [4] [5] [6] [9] [10] [39] [40] [42] [43] [68] How to think about agent frameworks
https://blog.langchain.com/how-to-think-about-agent-frameworks/

[7] [8] Workflows and agents - Docs by LangChain
https://docs.langchain.com/oss/python/langgraph/workflows-agents

[11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [48] [49] Build agentic systems with CrewAI and Amazon Bedrock | Artificial Intelligence
https://aws.amazon.com/blogs/machine-learning/build-agentic-systems-with-crewai-and-amazon-bedrock/

[25] [27] [28] [29] [30] [33] [34] [36] [37] [50] AutoGen - Microsoft Research
https://www.microsoft.com/en-us/research/project/autogen/

[26] [2308.08155] AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation
https://arxiv.org/abs/2308.08155

[31] [32] [35] AutoGen — AutoGen
https://microsoft.github.io/autogen/stable//index.html

[38] Build agents faster, your way - LangChain
https://www.langchain.com/langchain

41 51 69 LLM Powered Autonomous Agents | Lil'Log
https://lilianweng.github.io/posts/2023-06-23-agent/

44 45 46 47 [2311.17541] TaskWeaver: A Code-First Agent Framework
https://arxiv.org/abs/2311.17541

52 ReAct: Synergizing Reasoning and Acting in Language Models - Liner
https://liner.com/review/react-synergizing-reasoning-and-acting-in-language-models

53 ReAct: Combining Reasoning and Acting in Language Models for ...
https://medium.com/@LawrencewleKnight/react-combining-reasoning-and-acting-in-language-models-for-smarter-ai-66d844c1b499

54 55 [2303.17760] CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society
https://arxiv.org/abs/2303.17760

56 [2302.04761] Toolformer: Language Models Can Teach Themselves to Use Tools
https://arxiv.org/abs/2302.04761

57 58 61 [2303.11366] Reflexion: Language Agents with Verbal Reinforcement Learning
https://arxiv.org/abs/2303.11366

59 62 63 Built with LangGraph! #29: Reflection & Reflexion | by Okan Yenigün | Nov, 2025 | Towards Dev
https://towardsdev.com/built-with-langgraph-29-reflection-reflexion-10cc1cf96f35?gi=a4664e104299

60 Reflexion | Prompt Engineering Guide
https://www.promptingguide.ai/techniques/reflexion

64 65 66 67 74 [2507.22925] Hierarchical Memory for High-Efficiency Long-Term Reasoning in LLM Agents
https://arxiv.org/abs/2507.22925

70 71 72 User Frustration | Arize Docs
https://arize.com/docs/ax/evaluate/evaluators/llm-as-a-judge/arize-evaluators-llm-as-a-judge/user-frustration

73 LLM-as-a-judge: a complete guide to using LLMs for evaluations
https://www.evidentlyai.com/llm-guide/llm-as-a-judge

75 76 Optimierungsbedarf_AutoBot.txt
file://file_000000003ecc720abac52775d221aa53

77 78 Prompt Driven Development
https://capgemini.github.io/ai/prompt-driven-development/

79 80 90 91 92 93 94 Vibe Coding: The Art of New AI Prompt Style | by Soumava Dey | Medium
https://medium.com/@soumavadey/vibe-coding-the-art-of-new-ai-prompt-style-00e65dca25c6

81 82 83 84 85 87 88 89 95 96 97 98 99 100 101 Vibe Coding Prompting: Unlocking the Hidden Layer of AI Interaction | by Hex Shift | Medium
https://hexshift.medium.com/vibe-coding-prompting-unlocking-the-hidden-layer-of-ai-interaction-81c22dcbef9c

86 Shining a Light on “Shadow Prompting” | TechPolicy.Press
https://techpolicy.press/shining-a-light-on-shadow-prompting

102 The impact of generative AI on academic knowledge production in ...
https://policyreview.info/articles/news/what-alternative-impact-generative-ai-academic-knowledge-production-times-science

103 A Review on Vibe Coding: Fundamentals, State-of-the-art
https://www.techrxiv.org/users/913189/articles/1292402-a-review-on-vibe-coding-fundamentals-state-of-the-art-challenges-and-future-directions

104 Context engineering in agents - Docs by LangChain
https://docs.langchain.com/oss/python/langchain/context-engineering